

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_ARRAY_H
22: #define SPL_ARRAY_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26: #include "spl_iterators.h"
27:
28: extern ZEND_API zend_class_entry *spl_ce_ArrayObject;
29: extern ZEND_API zend_class_entry *spl_ce_ArrayIterator;
30: extern ZEND_API zend_class_entry *spl_ce_RecursiveArrayIterator;
31:
32: PHP_MINIT_FUNCTION(spl_array);
33:
34: extern void spl_array_iterator_append(zval *object, zval *append_value);
35: extern void spl_array_iterator_key(zval *object, zval *return_value);
36:
37: #endif /* SPL_ARRAY_H */
38:
39: /*
40:  * Local Variables:
41:  * c-basic-offset: 4
42:  * tab-width: 4
43:  * End:
44:  * vim600: fdm=marker
45:  * vim: noet sw=4 ts=4
46:  */
```

```
1: /*
2:  * -----
3:  * | PHP Version ? |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Stienne Kneuss <colder@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifdef HAVE_CONFIG_H
22: #include "config.h"
23: #endif
24:
25: #include "spl.h"
26: #include "zend_exceptions.h"
27:
28: #include "spl_heap.h"
29: #include "spl_functions.h"
30: #include "spl_engine.h"
31: #include "spl_iterators.h"
32: #include "spl_heap.h"
33: #include "spl_exceptions.h"
34:
35: #define PTR_HEAP_BLOCK_SIZE 64
36:
37: #define SPL_HEAP_CORRUPTED 0x00000001
38:
39: #define SPL_PRIORITY_EXTR_MARK 0x00000003
40: #define SPL_PRIORITY_EXTR_BOTH 0x00000003
41: #define SPL_PRIORITY_EXTR_DATA 0x00000001
42: #define SPL_PRIORITY_EXTR_PRIORITY 0x00000002
43:
44: zend_object_handlers spl_handler_SplHeap;
45: zend_object_handlers spl_handler_SplPriorityQueue;
46:
47: PHPAPI zend_class_entry *spl_ow_SplHeap;
48: PHPAPI zend_class_entry *spl_ow_SplMaxHeap;
49: PHPAPI zend_class_entry *spl_ow_SplMinHeap;
50: PHPAPI zend_class_entry *spl_ow_SplPriorityQueue;
51:
52:
53: typedef void (*spl_ptr_heap_dtor_func)(zval *);
54: typedef void (*spl_ptr_heap_ctor_func)(zval *);
55: typedef int (*spl_ptr_heap_cmp_func)(zval *, zval *, zval *);
56:
57: typedef struct _spl_ptr_heap {
58:     zval *elements;
59:     spl_ptr_heap_ctor_func ctor;
60:     spl_ptr_heap_dtor_func dtor;
61:     spl_ptr_heap_cmp_func cmp;
62:     int count;
63:     int max_size;
64:     int flags;
65: } spl_ptr_heap;
66:
67: typedef struct _spl_heap_object spl_heap_object;
68: typedef struct _spl_heap_it spl_heap_it;
69:
70: struct _spl_heap_object {
71:     spl_ptr_heap *heap;
72:     int flags;
73:     zend_class_entry *ce, spl_iterator;
74:     zend_function *fptr_cmp;
75:     zend_function *fptr_count;
76:     zend_object std;
77: };
78:
79: /* define an overloaded iterator structure */
80: struct _spl_heap_it {
81:     zend_user_iterator intern;
82:     int flags;
83: };
84:
85: static inline spl_heap_object *spl_heap_from_obj(zend_object *obj) /* {{{ */ {
86:     return (spl_heap_object*)((char*)obj - XOFFSOF(spl_heap_object, std));
87: } /* }}} */
88:
89:
90: #define Z_SPLHEAP_P(rv) spl_heap_from_obj(Z_OBJ_P(rv))
91:
92: static void spl_ptr_heap_dtor(zval *elem) /* {{{ */ {
93:     if (!IS_UNDEF_P(elem)) {
94:         spl_ptr_heap_dtor(elem);
95:     }
96: }
97: /* }}} */
98:
99: static void spl_ptr_heap_ctor(zval *elem) /* {{{ */ {
100:     Z_TRY_ADDREF_P(elem);
101: }
102: /* }}} */
103:
104: static int spl_ptr_heap_cmp_cb_helper(zval *obj, spl_heap_object *heap, zval *a, zval *b, zend_long *result) /* {{{ */ {
105:     zval result;
106:
107:     zend_call_method_with_2_params(obj, heap->std.ce, &heap->std.ce, "compare", &result, a, b);
108:
109:     if (EG(exception)) {
110:         return FAILURE;
111:     }
112:
113:     *result = zend_get_long(&result);
114:     spl_ptr_heap_dtor(&result);
115:
116:     return SUCCESS;
117: }
118: /* }}} */
119:
120: static zval *spl_pqueue_extract_helper(zval *value, int flags) /* {{{ */ {
121:     if ((flags & SPL_PRIORITY_EXTR_BOTH) == SPL_PRIORITY_EXTR_BOTH) {
122:         return value;
123:     } else if ((flags & SPL_PRIORITY_EXTR_BOTH) > 0) {
124:         if ((flags & SPL_PRIORITY_EXTR_DATA) == SPL_PRIORITY_EXTR_DATA) {
125:             zval *data;
126:             if ((data = zend_hash_str_find(Z_ARRVAL_P(value), "data", sizeof("data") - 1)) != NULL) {
127:                 return data;
128:             }
129:         } else {
130:             zval *priority;
131:             if ((priority = zend_hash_str_find(Z_ARRVAL_P(value), "priority", sizeof("priority") - 1)) != NULL) {
132:                 return priority;
133:             }
134:         }
135:     }
136:
137:     return NULL;
138: }
139: /* }}} */
140:
141: static int spl_ptr_heap_zval_max_cmp(zval *a, zval *b, zval *obj) /* {{{ */ {
142:     zval result;
143:
144:     if (EG(exception)) {
145:         return 0;
146:     }
147:
148:     if (obj) {
149:         spl_heap_object *heap_obj = Z_SPLHEAP_P(obj);
150:         if (heap_obj->fptr_cmp) {
151:             zend_long lval = 0;
152:             if (spl_ptr_heap_cmp_cb_helper(obj, heap_obj, a, b, &lval) == FAILURE) {
153:                 /* exception or call failure */
154:                 return 0;
155:             }
156:         }
157:         return lval > 0 ? 1 : (lval < 0 ? -1 : 0);
158:     }
159:
160:     compare_function(&result, a, b);
161:     return (int)Z_LVAL(result);
162: }
163: /* }}} */
164:
165: static int spl_ptr_heap_zval_min_cmp(zval *a, zval *b, zval *obj) /* {{{ */ {
166:     zval result;
167:
168:     if (EG(exception)) {
169:         return 0;
170:     }
171:
172:     if (obj) {
173:         spl_heap_object *heap_obj = Z_SPLHEAP_P(obj);
174:         if (heap_obj->fptr_cmp) {
175:             zend_long lval = 0;
176:             if (spl_ptr_heap_cmp_cb_helper(obj, heap_obj, a, b, &lval) == FAILURE) {
177:                 /* exception or call failure */
178:                 return 0;
179:             }
180:         }
181:         return lval > 0 ? 1 : (lval < 0 ? -1 : 0);
182:     }
183:
184:     compare_function(&result, b, a);
185:     return (int)Z_LVAL(result);
186: }
187: /* }}} */

```

```
188:
189: static int spl_ptr_pqueue_zval_cmp(zval *a, zval *b, zval *obj) /* {{{ */ {
190:     zval result;
191:     zval *a_priority_p = spl_pqueue_extract_helper(a, SPL_PRIORITY_EXTR_PRIORITY);
192:     zval *b_priority_p = spl_pqueue_extract_helper(b, SPL_PRIORITY_EXTR_PRIORITY);
193:
194:     if (!(!a_priority_p || !b_priority_p)) {
195:         zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
196:         return 0;
197:     }
198:
199:     if (EG(exception)) {
200:         return 0;
201:     }
202:
203:     if (obj) {
204:         spl_heap_object *heap_obj = Z_SPLHEAP_P(obj);
205:         if (heap_obj->fptr_cmp) {
206:             zend_long lval = 0;
207:             if (spl_ptr_heap_cmp_cb_helper(obj, heap_obj, a_priority_p, b_priority_p, &lval) == FAILURE) {
208:                 /* exception or call failure */
209:                 return 0;
210:             }
211:         }
212:         return lval > 0 ? 1 : (lval < 0 ? -1 : 0);
213:     }
214:
215:     compare_function(&result, a_priority_p, b_priority_p);
216:     return (int)Z_LVAL(result);
217: }
218: /* }}} */
219:
220: static spl_ptr_heap *spl_ptr_heap_init(spl_ptr_heap_cmp_func cmp, spl_ptr_heap_ctor_func ctor, spl_ptr_heap_dtor_func dtor) /* {{{ */ {
221:     spl_ptr_heap *heap = emalloc(sizeof(spl_ptr_heap));
222:
223:     heap->dtor = dtor;
224:     heap->ctor = ctor;
225:     heap->cmp = cmp;
226:     heap->elements = ecalloc(PTR_HEAP_BLOCK_SIZE, sizeof(zval));
227:     heap->max_size = PTR_HEAP_BLOCK_SIZE;
228:     heap->count = 0;
229:     heap->flags = 0;
230:
231:     return heap;
232: }
233: /* }}} */
234:
235: static void spl_ptr_heap_insert(spl_ptr_heap *heap, zval *elem, void *cmp_userdata) /* {{{ */ {
236:     int i;
237:
238:     if (heap->count+1 > heap->max_size) {
239:         /* we need to allocate more memory */
240:         heap->elements = erealloc(heap->elements, heap->max_size * 2 * sizeof(zval));
241:         memset(heap->elements + heap->max_size, 0, heap->max_size * sizeof(zval));
242:         heap->max_size *= 2;
243:     }
244:
245:     /* sifting up */
246:     for (i = heap->count; i > 0 & heap->cmp(heap->elements[(i-1)/2], elem, cmp_userdata) < 0; i = (i-1)/2) {
247:         heap->elements[i] = heap->elements[(i-1)/2];
248:     }
249:     heap->count++;
250:
251:     if (EG(exception)) {
252:         /* exception thrown during comparison */
253:         heap->flags |= SPL_HEAP_CORRUPTED;
254:     }
255:
256:     ZVAL_COPY_VALUE(heap->elements[i], elem);
257:
258:     /* sifting up */
259:     static zval *spl_ptr_heap_top(spl_ptr_heap *heap) /* {{{ */ {
260:         if (heap->count == 0) {
261:             return NULL;
262:         }
263:         return Z_UNDEF(heap->elements[0]) ? NULL : heap->elements[0];
264:     }
265:
266:     static void spl_ptr_heap_delete_top(spl_ptr_heap *heap, zval *elem, void *cmp_userdata) /* {{{ */ {
267:         int i, j;
268:         zend_long limit = (heap->count-1)/2;
269:         zval *bottom;
270:
271:         if (heap->count == 0) {
272:             ZVAL_UNDEF(elem);
273:             return;
274:         }
275:
276:         ZVAL_COPY_VALUE(elem, heap->elements[0]);
277:         bottom = heap->elements[--heap->count];
278:
279:         for (i = 0; i < limit; i++) {
280:             /* find smaller child */
281:             j = i * 2 + 1;
282:             if (j > heap->count & heap->cmp(heap->elements[j], heap->elements[j], cmp_userdata) > 0) {
283:                 /* next child is bigger */
284:                 break;
285:             }
286:
287:             /* swap elements between two levels */
288:             if (heap->cmp(bottom, heap->elements[j], cmp_userdata) < 0) {
289:                 heap->elements[i] = heap->elements[j];
290:             } else {
291:                 break;
292:             }
293:
294:             if (EG(exception)) {
295:                 /* exception thrown during comparison */
296:                 heap->flags |= SPL_HEAP_CORRUPTED;
297:             }
298:
299:             ZVAL_COPY_VALUE(heap->elements[i], bottom);
300:         }
301:
302:         /* sifting up */
303:         static spl_ptr_heap_clone(spl_ptr_heap *from) /* {{{ */ {
304:             int i;
305:
306:             spl_ptr_heap *heap = emalloc(sizeof(spl_ptr_heap));
307:
308:             heap->dtor = from->dtor;
309:             heap->ctor = from->ctor;
310:             heap->cmp = from->cmp;
311:             heap->max_size = from->max_size;
312:             heap->count = from->count;
313:             heap->flags = from->flags;
314:
315:             heap->elements = safe_emalloc(sizeof(zval), from->max_size, 0);
316:             memcpy(heap->elements, from->elements, sizeof(zval)*from->max_size);
317:
318:             for (i=0; i < heap->count; ++i) {
319:                 heap->elements[i] = from->elements[i];
320:             }
321:
322:             return heap;
323:         }
324:
325:         static void spl_ptr_heap_destroy(spl_ptr_heap *heap) /* {{{ */ {
326:             int i;
327:
328:             for (i=0; i < heap->count; ++i) {
329:                 heap->dtor(heap->elements[i]);
330:             }
331:
332:             zfree(heap->elements);
333:             zfree(heap);
334:         }
335:
336:         static int spl_ptr_heap_count(spl_ptr_heap *heap) /* {{{ */ {
337:             return heap->count;
338:         }
339:
340:         static void spl_ptr_heap_get_iterator(zend_class_entry *ce, zval *obj, int by_ref) {
341:             spl_ptr_heap_object *intern = spl_heap_from_obj(obj);
342:
343:             zend_object_std_init(&intern->std, ce);
344:             spl_ptr_heap_destroy(&intern->heap);
345:
346:             /* sifting up */
347:
348:             static zend_object *spl_heap_object_new_ex(zend_class_entry *class_type, zval *orig, int clone_orig) /* {{{ */ {
349:                 spl_ptr_heap_object *intern;
350:                 zend_class_entry *parent = class_type;
351:                 int inherited = 0;
352:
353:                 intern = zend_object_alloc(sizeof(spl_ptr_heap_object), parent);
354:
355:                 zend_object_std_init(&intern->std, class_type);
356:                 object_properties_init(&intern->std, class_type);
357:
358:                 intern->flags = 0;
359:                 intern->fptr_cmp = NULL;
360:
361:                 if (orig) {
362:                     spl_ptr_heap_get_iterator(class_type, orig, 0);
363:                     intern->ce_get_iterator = orig->ce_get_iterator;
364:                 }
365:             }
366:         }
367:     }
368: }
369: /* }}} */

```

```

377:
378:     IF (clone_orig) {
379:         intern->heap = spl_ptr_heap_clone(other->heap);
380:     } else {
381:         intern->heap = other->heap;
382:     }
383:
384:     intern->flags = other->flags;
385: } else {
386:     intern->heap = spl_ptr_heap_init(spl_ptr_heap_eval_max_cmp, spl_ptr_heap_eval_ctor, spl_ptr_heap_eval_dtor);
387: }
388:
389: intern->std.handlers = spl_handler_SplHeap;
390:
391: while (parent) {
392:     IF (parent == spl_ce_SplPriorityQueue) {
393:         intern->heap->cmp = spl_ptr_pqueue_eval_cmp;
394:         intern->flags = SPL_PRIORITY_QUEUE_EXTRA_DATA;
395:         intern->std.handlers = spl_handler_SplPriorityQueue;
396:         break;
397:     }
398:
399:     IF (parent == spl_ce_SplMinHeap) {
400:         intern->heap->cmp = spl_ptr_heap_eval_min_cmp;
401:         break;
402:     }
403:
404:     IF (parent == spl_ce_SplMaxHeap) {
405:         intern->heap->cmp = spl_ptr_heap_eval_max_cmp;
406:         break;
407:     }
408:
409:     IF (parent == spl_ce_SplHeap) {
410:         break;
411:     }
412:
413:     parent = parent->parent;
414:     inherited = 1;
415: }
416:
417: IF (!parent) { /* this must never happen */
418:     php_error_docref(NULL, E_COMPILE_ERROR, "Internal compiler error, class is not child of SplHeap");
419: }
420:
421: IF (inherited) {
422:     intern->fptr_cmp = zend_hash_str_find_ptr(class_type->function_table, "compare", sizeof("compare") - 1);
423:     IF (intern->fptr_cmp->common.scope == parent) {
424:         intern->fptr_cmp = NULL;
425:     }
426:     intern->fptr_count = zend_hash_str_find_ptr(class_type->function_table, "count", sizeof("count") - 1);
427:     IF (intern->fptr_count->common.scope == parent) {
428:         intern->fptr_count = NULL;
429:     }
430: }
431:
432: return sintern->std;
433: } /* }}} */
434:
435:
436: static zend_object *spl_heap_object_new(zend_class_entry *class_type) /* {{{ */
437: {
438:     return spl_heap_object_new_ex(class_type, NULL, 0);
439: }
440: /* }}} */
441:
442: static zend_object *spl_heap_object_clone(zval *obj) /* {{{ */
443: {
444:     zend_object *old_obj;
445:     zend_object *new_obj;
446:
447:     old_obj = Z_OBJ_P(obj);
448:     new_obj = spl_heap_object_new_ex(old_obj->ce, obj, 1);
449:
450:     zend_objects_clone_members(new_obj, old_obj);
451:
452:     return new_obj;
453: }
454: /* }}} */
455:
456: static int spl_heap_object_count_elements(zval *obj, zend_long *count) /* {{{ */
457: {
458:     spl_heap_object *intern = Z_SPLHEAP_P(obj);
459:
460:     IF (intern->fptr_count) {
461:         zval rv;
462:         zend_call_method_with_0_params(object, intern->std.ce, sintern->fptr_count, "count", &rv);
463:         IF (!Z_ISUNDEF(rv)) {
464:             *count = zval_get_long(&rv);
465:             zval_ptr_dtor(&rv);
466:             return SUCCESS;
467:         }
468:         *count = 0;
469:         return FAILURE;
470:     }
471:
472:     *count = spl_ptr_heap_count(intern->heap);
473:
474:     return SUCCESS;
475: }
476: /* }}} */
477:
478: static HashTable *spl_heap_object_get_debug_info_helper(zend_class_entry *ce, zval *obj, int *is_temp) /* {{{ */
479: {
480:     spl_heap_object *intern = Z_SPLHEAP_P(obj);
481:     zval tmp_heap_array;
482:     zend_string *pnstr;
483:     HashTable *debug_info;
484:     int i;
485:
486:     *is_temp = 1;
487:
488:     IF (!intern->std.properties) {
489:         rebuild_object_properties(intern->std);
490:     }
491:
492:     debug_info = zend_new_array(zend_hash_num_elements(intern->std.properties) + 1);
493:     zend_hash_copy(debug_info, intern->std.properties, (copy_ctor_func_t) zval_add_ref);
494:
495:     pnstr = spl_gen_private_prop_name(ce, "Flags", sizeof("Flags") - 1);
496:     ZVAL_LONG(&tmp_heap_array, intern->flags);
497:     zend_hash_update(debug_info, pnstr, &tmp_heap_array);
498:     zend_string_release(pnstr);
499:
500:     pnstr = spl_gen_private_prop_name(ce, "IsCorrupted", sizeof("IsCorrupted") - 1);
501:     ZVAL_BOOL(&tmp_heap_array, intern->heap->flags & SPL_HEAP_CORRUPTED);
502:     zend_hash_update(debug_info, pnstr, &tmp_heap_array);
503:     zend_string_release(pnstr);
504:
505:     array_init(&debug_info);
506:
507:     for (i = 0; i < intern->heap->count; ++i) {
508:         add_index_zval(&debug_info, i, sintern->heap->elements[i]);
509:         IF (Z_REFCOUNTED(intern->heap->elements[i])) {
510:             Z_ADDREF(intern->heap->elements[i]);
511:         }
512:     }
513:
514:     pnstr = spl_gen_private_prop_name(ce, "Heap", sizeof("Heap") - 1);
515:     zend_hash_update(debug_info, pnstr, &debug_info);
516:     zend_string_release(pnstr);
517:
518:     return debug_info;
519: } /* }}} */
520:
521: static HashTable *spl_heap_object_get_ce(zval *obj, zval **gc_data, int *gc_data_count) /* {{{ */
522: {
523:     spl_heap_object *intern = Z_SPLHEAP_P(obj);
524:     *gc_data = intern->heap->elements[0];
525:     *gc_data_count = intern->heap->count;
526:
527:     return std_object_handlers.get_properties(obj);
528: }
529: /* }}} */
530:
531: static HashTable *spl_heap_object_get_debug_info(zval *obj, int *is_temp) /* {{{ */
532: {
533:     return spl_heap_object_get_debug_info_helper(spl_ce_SplHeap, obj, is_temp);
534: }
535: /* }}} */
536:
537: static HashTable *spl_pqueue_object_get_debug_info(zval *obj, int *is_temp) /* {{{ */
538: {
539:     return spl_heap_object_get_debug_info_helper(spl_ce_SplPriorityQueue, obj, is_temp);
540: }
541: /* }}} */
542:
543: /* {{{ proto int SplHeap::count()
544: Return the number of elements in the heap. */
545: SPL_METHOD(SplHeap, count)
546: {
547:     zend_long count;
548:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
549:
550:     IF (zend_parse_parameters_none() == FAILURE) {
551:         return;
552:     }
553:
554:     count = spl_ptr_heap_count(intern->heap);
555:     RETURN_LONG(count);
556: }
557: /* }}} */
558:
559: /* {{{ proto int SplHeap::isEmpty()
560: Return true if the heap is empty. */
561: SPL_METHOD(SplHeap, isEmpty)
562: {
563:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
564:

```

```

565:     IF (zend_parse_parameters_none() == FAILURE) {
566:         return;
567:     }
568:
569:     RETURN_BOOL(spl_ptr_heap_count(intern->heap) == 0);
570: }
571: /* }}} */
572:
573: /* {{{ proto bool SplHeap::insert(mixed value)
574: Push $value on the heap */
575: SPL_METHOD(SplHeap, insert)
576: {
577:     zval *value;
578:     spl_heap_object *intern;
579:
580:     IF (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &value) == FAILURE) {
581:         return;
582:     }
583:
584:     intern = Z_SPLHEAP_P(getThis());
585:
586:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
587:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
588:         return;
589:     }
590:
591:     Z_TRY_ADDREF_P(value);
592:     spl_ptr_heap_insert(intern->heap, value, getThis());
593:
594:     RETURN_TRUE;
595: }
596: /* }}} */
597:
598: /* {{{ proto mixed SplHeap::extract()
599: Extract the element out of the top of the heap */
600: SPL_METHOD(SplHeap, extract)
601: {
602:     spl_heap_object *intern;
603:
604:     IF (zend_parse_parameters_none() == FAILURE) {
605:         return;
606:     }
607:
608:     intern = Z_SPLHEAP_P(getThis());
609:
610:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
611:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
612:         return;
613:     }
614:
615:     spl_ptr_heap_delete_top(intern->heap, return_value, getThis());
616:
617:     IF (Z_ISUNDEF_P(return_value)) {
618:         zend_throw_exception(spl_ce_RuntimeException, "Can't extract from an empty heap", 0);
619:         return;
620:     }
621: }
622: /* }}} */
623:
624: /* {{{ proto bool SplPriorityQueue::insert(mixed value, mixed priority)
625: Push $value with the priority $priority on the priorityqueue */
626: SPL_METHOD(SplPriorityQueue, insert)
627: {
628:     zval *data, *priority, elem;
629:     spl_heap_object *intern;
630:
631:     IF (zend_parse_parameters(ZEND_NUM_ARGS(), "as", &data, &priority) == FAILURE) {
632:         return;
633:     }
634:
635:     intern = Z_SPLHEAP_P(getThis());
636:
637:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
638:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
639:         return;
640:     }
641:
642:     Z_TRY_ADDREF_P(data);
643:     Z_TRY_ADDREF_P(priority);
644:
645:     array_init(&elem);
646:     add_assoc_zval_ex(&elem, "data", sizeof("data") - 1, data);
647:     add_assoc_zval_ex(&elem, "priority", sizeof("priority") - 1, priority);
648:
649:     spl_ptr_heap_insert(intern->heap, &elem, getThis());
650:
651:     RETURN_TRUE;
652: }
653: /* }}} */
654:
655: /* {{{ proto mixed SplPriorityQueue::extract()
656: Extract the element out of the top of the priority queue */
657: SPL_METHOD(SplPriorityQueue, extract)
658: {
659:     zval value, *value_out;
660:     spl_heap_object *intern;
661:
662:     IF (zend_parse_parameters_none() == FAILURE) {
663:         return;
664:     }
665:
666:     intern = Z_SPLHEAP_P(getThis());
667:
668:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
669:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
670:         return;
671:     }
672:
673:     spl_ptr_heap_delete_top(intern->heap, &value, getThis());
674:
675:     IF (Z_ISUNDEF(value)) {
676:         zend_throw_exception(spl_ce_RuntimeException, "Can't extract from an empty heap", 0);
677:         return;
678:     }
679:
680:     value_out = spl_pqueue_extract_helper(&value, intern->flags);
681:
682:     IF (!value_out) {
683:         zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
684:         zval_ptr_dtor(&value);
685:         return;
686:     }
687:
688:     ZVAL_DEREF(value_out);
689:     ZVAL_COPY(&return_value, value_out);
690:     zval_ptr_dtor(&value);
691: }
692: /* }}} */
693:
694: /* {{{ proto mixed SplPriorityQueue::top()
695: Peek at the top element of the priority queue */
696: SPL_METHOD(SplPriorityQueue, top)
697: {
698:     zval *value, *value_out;
699:     spl_heap_object *intern;
700:
701:     IF (zend_parse_parameters_none() == FAILURE) {
702:         return;
703:     }
704:
705:     intern = Z_SPLHEAP_P(getThis());
706:
707:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
708:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
709:         return;
710:     }
711:
712:     value = spl_ptr_heap_top(intern->heap);
713:
714:     IF (!value) {
715:         zend_throw_exception(spl_ce_RuntimeException, "Can't peek at an empty heap", 0);
716:         return;
717:     }
718:
719:     value_out = spl_pqueue_extract_helper(value, intern->flags);
720:
721:     IF (!value_out) {
722:         zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
723:         return;
724:     }
725:
726:     ZVAL_DEREF(value_out);
727:     ZVAL_COPY(&return_value, value_out);
728: }
729: /* }}} */
730:
731: /* {{{ proto int SplPriorityQueue::isotExtractFlags(int flags)
732: Get the flags of extraction */
733: SPL_METHOD(SplPriorityQueue, isotExtractFlags)
734: {
735:     zend_long value;
736:     spl_heap_object *intern;
737:
738:     IF (zend_parse_parameters(ZEND_NUM_ARGS(), "i", &value) == FAILURE) {
739:         return;
740:     }
741:
742:     intern = Z_SPLHEAP_P(getThis());
743:
744:     intern->flags = value & SPL_PRIORITY_QUEUE_MASK;
745:
746:     RETURN_LONG(intern->flags);
747: }
748:
749: /* }}} */
750:
751: /* {{{ proto int SplPriorityQueue::getExtractFlags()
752: Get the flags of extraction */

```

```

753: SPL_METHOD(SplPriorityQueue, getExtractFlags)
754: {
755:     spl_heap_object *intern;
756:
757:     IF (zend_parse_parameters_none() == FAILURE) {
758:         return;
759:     }
760:
761:     intern = Z_SPLHEAP_P(getThis());
762:
763:     RETURN_LONG(intern->heap->flags);
764: }
765: /* }}} */
766:
767: /* {{{ proto int SplHeap::recoverFromCorruption()
768:  * Recover from a corrupted state */
769: SPL_METHOD(SplHeap, recoverFromCorruption)
770: {
771:     spl_heap_object *intern;
772:
773:     IF (zend_parse_parameters_none() == FAILURE) {
774:         return;
775:     }
776:
777:     intern = Z_SPLHEAP_P(getThis());
778:
779:     intern->heap->flags = intern->heap->flags & SPL_HEAP_CORRUPTED;
780:
781:     RETURN_TRUE;
782: }
783: /* }}} */
784:
785: /* {{{ proto int SplHeap::isCorrupted()
786:  * Tells if the heap is in a corrupted state */
787: SPL_METHOD(SplHeap, isCorrupted)
788: {
789:     spl_heap_object *intern;
790:
791:     IF (zend_parse_parameters_none() == FAILURE) {
792:         return;
793:     }
794:
795:     intern = Z_SPLHEAP_P(getThis());
796:
797:     RETURN_BOOL(intern->heap->flags & SPL_HEAP_CORRUPTED);
798: }
799: /* }}} */
800:
801: /* {{{ proto bool SplPriorityQueue::compare(mixed $a, mixed $b)
802:  * compare the priorities */
803: SPL_METHOD(SplPriorityQueue, compare)
804: {
805:     zval *a, *b;
806:
807:     IF (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "zz", &a, &b) == FAILURE) {
808:         return;
809:     }
810:
811:     RETURN_LONG(spl_ptr_heap_zval_max_cmp(a, b, NULL));
812: }
813: /* }}} */
814:
815: /* {{{ proto mixed SplHeap::top()
816:  * Peek at the top element of the heap */
817: SPL_METHOD(SplHeap, top)
818: {
819:     zval *value;
820:     spl_heap_object *intern;
821:
822:     IF (zend_parse_parameters_none() == FAILURE) {
823:         return;
824:     }
825:
826:     intern = Z_SPLHEAP_P(getThis());
827:
828:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
829:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
830:         return;
831:     }
832:
833:     value = spl_ptr_heap_top(intern->heap);
834:
835:     IF (!value) {
836:         zend_throw_exception(spl_ce_RuntimeException, "Can't peek at an empty heap", 0);
837:         return;
838:     }
839:
840:     ZVAL_DEREF(value);
841:     ZVAL_COPY(return_value, value);
842: }
843: /* }}} */
844:
845: /* {{{ proto bool SplMinHeap::compare(mixed $a, mixed $b)
846:  * compare the values */
847: SPL_METHOD(SplMinHeap, compare)
848: {
849:     zval *a, *b;
850:
851:     IF (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "zz", &a, &b) == FAILURE) {
852:         return;
853:     }
854:
855:     RETURN_LONG(spl_ptr_heap_zval_min_cmp(a, b, NULL));
856: }
857: /* }}} */
858:
859: /* {{{ proto bool SplMaxHeap::compare(mixed $a, mixed $b)
860:  * compare the values */
861: SPL_METHOD(SplMaxHeap, compare)
862: {
863:     zval *a, *b;
864:
865:     IF (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "zz", &a, &b) == FAILURE) {
866:         return;
867:     }
868:
869:     RETURN_LONG(spl_ptr_heap_zval_max_cmp(a, b, NULL));
870: }
871: /* }}} */
872:
873: static void spl_heap_it_dtor(zend_object_iterator *iter) /* {{{ */
874: {
875:     spl_heap_it *iterator = (spl_heap_it *)iter;
876:
877:     zend_user_it_invalidata_current(iter);
878:     zval_ptr_dtor(iterator->intern.it.data);
879: }
880: /* }}} */
881:
882: static void spl_heap_it_rewind(zend_object_iterator *iter) /* {{{ */
883: {
884:     /* do nothing, the iterator always points to the top element */
885: }
886: /* }}} */
887:
888: static int spl_heap_it_valid(zend_object_iterator *iter) /* {{{ */
889: {
890:     return ((Z_SPLHEAP_P(iter->data)->heap->count != 0 ? SUCCESS : FAILURE);
891: }
892: /* }}} */
893:
894: static zval *spl_heap_it_get_current_data(zend_object_iterator *iter) /* {{{ */
895: {
896:     spl_heap_object *object = Z_SPLHEAP_P(iter->data);
897:     zval *element = object->heap->elements[0];
898:
899:     IF (object->heap->flags & SPL_HEAP_CORRUPTED) {
900:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
901:         return NULL;
902:     }
903:
904:     IF (object->heap->count == 0 || Z_UNDEF_P(element)) {
905:         return NULL;
906:     } else {
907:         return element;
908:     }
909: }
910: /* }}} */
911:
912: static zval *spl_pqueue_it_get_current_data(zend_object_iterator *iter) /* {{{ */
913: {
914:     spl_heap_object *object = Z_SPLHEAP_P(iter->data);
915:     zval *element = object->heap->elements[0];
916:
917:     IF (object->heap->flags & SPL_HEAP_CORRUPTED) {
918:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
919:         return NULL;
920:     }
921:
922:     IF (object->heap->count == 0 || Z_UNDEF_P(element)) {
923:         return NULL;
924:     } else {
925:         zval *data = spl_pqueue_extract_helper(element, object->flags);
926:         IF (!data) {
927:             zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
928:         }
929:         return data;
930:     }
931: }
932: /* }}} */
933:
934: static void spl_heap_it_get_current_key(zend_object_iterator *iter, zval **key) /* {{{ */
935: {
936:     spl_heap_object *object = Z_SPLHEAP_P(iter->data);
937:
938:     ZVAL_LONG(key, object->heap->count - 1);
939: }
940: /* }}} */

```

```

941:
942: static void spl_heap_it_move_forward(zend_object_iterator *iter) /* {{{ */
943: {
944:     spl_heap_object *object = Z_SPLHEAP_P(iter->data);
945:     zval elem;
946:
947:     IF (object->heap->flags & SPL_HEAP_CORRUPTED) {
948:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
949:         return;
950:     }
951:
952:     spl_ptr_heap_delete_top(object->heap, &elem, iter->data);
953:
954:     zval_ptr_dtor(&elem);
955:
956:     zend_user_it_invalidata_current(iter);
957: }
958: /* }}} */
959:
960: /* {{{ proto int SplHeap::key()
961:  * Return current array key */
962: SPL_METHOD(SplHeap, key)
963: {
964:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
965:
966:     IF (zend_parse_parameters_none() == FAILURE) {
967:         return;
968:     }
969:
970:     RETURN_LONG(intern->heap->count - 1);
971: }
972: /* }}} */
973:
974: /* {{{ proto void SplHeap::next()
975:  * Move to next entry */
976: SPL_METHOD(SplHeap, next)
977: {
978:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
979:     zval elem;
980:     spl_ptr_heap_delete_top(intern->heap, &elem, getThis());
981:
982:     IF (zend_parse_parameters_none() == FAILURE) {
983:         return;
984:     }
985:
986:     zval_ptr_dtor(&elem);
987: }
988: /* }}} */
989:
990: /* {{{ proto bool SplHeap::valid()
991:  * Check whether the datastructure contains more entries */
992: SPL_METHOD(SplHeap, valid)
993: {
994:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
995:
996:     IF (zend_parse_parameters_none() == FAILURE) {
997:         return;
998:     }
999:
1000:     RETURN_BOOL(intern->heap->count != 0);
1001: }
1002: /* }}} */
1003:
1004: /* {{{ proto void SplHeap::rewind()
1005:  * Rewind the datastructure back to the start */
1006: SPL_METHOD(SplHeap, rewind)
1007: {
1008:     IF (zend_parse_parameters_none() == FAILURE) {
1009:         return;
1010:     }
1011:     /* do nothing, the iterator always points to the top element */
1012: }
1013: /* }}} */
1014:
1015: /* {{{ proto mixed NULL SplHeap::current()
1016:  * Return current datastructure entry */
1017: SPL_METHOD(SplHeap, current)
1018: {
1019:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
1020:     zval *element = intern->heap->elements[0];
1021:
1022:     IF (zend_parse_parameters_none() == FAILURE) {
1023:         return;
1024:     }
1025:
1026:     IF (!intern->heap->count || Z_UNDEF_P(element)) {
1027:         RETURN_NULL();
1028:     } else {
1029:         ZVAL_DEREF(element);
1030:         ZVAL_COPY(return_value, element);
1031:     }
1032: }
1033: /* }}} */
1034:
1035: /* {{{ proto mixed NULL SplPriorityQueue::current()
1036:  * Return current datastructure entry */
1037: SPL_METHOD(SplPriorityQueue, current)
1038: {
1039:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
1040:     zval *element = intern->heap->elements[0];
1041:
1042:     IF (zend_parse_parameters_none() == FAILURE) {
1043:         return;
1044:     }
1045:
1046:     IF (!intern->heap->count || Z_UNDEF_P(element)) {
1047:         RETURN_NULL();
1048:     } else {
1049:         zval *data = spl_pqueue_extract_helper(element, intern->flags);
1050:
1051:         IF (!data) {
1052:             zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
1053:             RETURN_NULL();
1054:         }
1055:
1056:         ZVAL_DEREF(data);
1057:         ZVAL_COPY(return_value, data);
1058:     }
1059: }
1060: /* }}} */
1061:
1062: /* Iterator handler table */
1063: static const zend_object_iterator_funcs spl_heap_it_funcs = {
1064:     spl_heap_it_dtor,
1065:     spl_heap_it_valid,
1066:     spl_heap_it_get_current_data,
1067:     spl_heap_it_get_current_key,
1068:     spl_heap_it_move_forward,
1069:     spl_heap_it_rewind,
1070:     NULL
1071: };
1072:
1073: static const zend_object_iterator_funcs spl_pqueue_it_funcs = {
1074:     spl_heap_it_dtor,
1075:     spl_heap_it_valid,
1076:     spl_pqueue_it_get_current_data,
1077:     spl_heap_it_get_current_key,
1078:     spl_heap_it_move_forward,
1079:     spl_heap_it_rewind,
1080:     NULL
1081: };
1082:
1083: zend_object_iterator *spl_heap_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
1084: {
1085:     spl_heap_it *iterator;
1086:     spl_heap_object *heap_object = Z_SPLHEAP_P(object);
1087:
1088:     IF (by_ref) {
1089:         zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1090:         return NULL;
1091:     }
1092:
1093:     iterator = emalloc(sizeof(spl_heap_it));
1094:
1095:     zend_iterator_init(iterator->intern.it);
1096:
1097:     ZVAL_COPY(iterator->intern.it.data, object);
1098:     iterator->intern.it.funcs = &spl_heap_it_funcs;
1099:     iterator->intern.ce = ce;
1100:     iterator->flags = heap_object->flags;
1101:     ZVAL_UNDEF(iterator->intern.value);
1102:
1103:     return iterator->intern.it;
1104: }
1105: /* }}} */
1106:
1107: zend_object_iterator *spl_pqueue_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
1108: {
1109:     spl_heap_it *iterator;
1110:     spl_heap_object *heap_object = Z_SPLHEAP_P(object);
1111:
1112:     IF (by_ref) {
1113:         zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1114:         return NULL;
1115:     }
1116:
1117:     iterator = emalloc(sizeof(spl_heap_it));
1118:
1119:     zend_iterator_init(&zend_object_iterator_iterator);
1120:
1121:     ZVAL_COPY(iterator->intern.it.data, object);
1122:     iterator->intern.it.funcs = &spl_pqueue_it_funcs;
1123:     iterator->intern.ce = ce;
1124:     iterator->flags = heap_object->flags;
1125:
1126:     ZVAL_UNDEF(iterator->intern.value);
1127:
1128:     return iterator->intern.it;

```

```
1129: }
1130: /* }}} */
1131:
1132: ZEND_BEGIN_ARG_INFO(arginfo_heap_insert, 0)
1133: ZEND_ARG_INFO(0, value)
1134: ZEND_END_ARG_INFO()
1135:
1136: ZEND_BEGIN_ARG_INFO(arginfo_heap_compare, 0)
1137: ZEND_ARG_INFO(0, a)
1138: ZEND_ARG_INFO(0, b)
1139: ZEND_END_ARG_INFO()
1140:
1141: ZEND_BEGIN_ARG_INFO(arginfo_pqqueue_insert, 0)
1142: ZEND_ARG_INFO(0, value)
1143: ZEND_ARG_INFO(0, priority)
1144: ZEND_END_ARG_INFO()
1145:
1146: ZEND_BEGIN_ARG_INFO(arginfo_pqqueue_setflags, 0)
1147: ZEND_ARG_INFO(0, flags)
1148: ZEND_END_ARG_INFO()
1149:
1150: ZEND_BEGIN_ARG_INFO(arginfo_splheap_void, 0)
1151: ZEND_END_ARG_INFO()
1152:
1153: static const zend_function_entry spl_funcs_SplMinHeap[] = {
1154:     SPL_ME(SplMinHeap, compare, arginfo_heap_compare, ZEND_ACC_PROTECTED)
1155:     PHP_FE_END
1156: };
1157:
1158: static const zend_function_entry spl_funcs_SplMaxHeap[] = {
1159:     SPL_ME(SplMaxHeap, compare, arginfo_heap_compare, ZEND_ACC_PROTECTED)
1160:     PHP_FE_END
1161: };
1162:
1163: static const zend_function_entry spl_funcs_SplPriorityQueue[] = {
1164:     SPL_ME(SplPriorityQueue, compare, arginfo_heap_compare, ZEND_ACC_PUBLIC)
1165:     SPL_ME(SplPriorityQueue, insert, arginfo_pqqueue_insert, ZEND_ACC_PUBLIC)
1166:     SPL_ME(SplPriorityQueue, setExtractFlags, arginfo_pqqueue_setflags, ZEND_ACC_PUBLIC)
1167:     SPL_ME(SplPriorityQueue, getExtractFlags, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1168:     SPL_ME(SplPriorityQueue, top, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1169:     SPL_ME(SplPriorityQueue, extract, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1170:     SPL_ME(SplHeap, count, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1171:     SPL_ME(SplHeap, isEmpty, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1172:     SPL_ME(SplPriorityQueue, current, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1173:     SPL_ME(SplHeap, key, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1174:     SPL_ME(SplHeap, rewind, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1175:     SPL_ME(SplHeap, valid, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1176:     SPL_ME(SplHeap, recoverFromCorruption, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1177:     SPL_ME(SplHeap, isCorrupted, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1178:     PHP_FE_END
1179: };
1180:
1181: static const zend_function_entry spl_funcs_SplHeap[] = {
1182:     SPL_ME(SplHeap, extract, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1183:     SPL_ME(SplHeap, insert, arginfo_heap_insert, ZEND_ACC_PUBLIC)
1184:     SPL_ME(SplHeap, top, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1185:     SPL_ME(SplHeap, count, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1186:     SPL_ME(SplHeap, isEmpty, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1187:     SPL_ME(SplHeap, rewind, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1188:     SPL_ME(SplHeap, current, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1189:     SPL_ME(SplHeap, key, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1190:     SPL_ME(SplHeap, next, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1191:     SPL_ME(SplHeap, valid, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1192:     SPL_ME(SplHeap, recoverFromCorruption, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1193:     SPL_ME(SplHeap, isCorrupted, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1194:     ZEND_FENTRY(compare, NULL, NULL, ZEND_ACC_PROTECTED|ZEND_ACC_ABSTRACT)
1195:     PHP_FE_END
1196: };
1197: /* }}} */
1198:
1199: PHP_MINIT_FUNCTION(spl_heap) /* {{{ */
1200: {
1201:     REGISTER_SPL_STD_CLASS_EX(SplHeap, spl_heap_object_new, spl_funcs_SplHeap);
1202:     memcpy(&spl_handler_SplHeap, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1203:
1204:     spl_handler_SplHeap.offset = XOffsetOf(spl_heap_object, std);
1205:     spl_handler_SplHeap.clone_obj = spl_heap_object_clone;
1206:     spl_handler_SplHeap.count_elements = spl_heap_object_count_elements;
1207:     spl_handler_SplHeap.get_debug_info = spl_heap_object_get_debug_info;
1208:     spl_handler_SplHeap.get_gc = spl_heap_object_get_gc;
1209:     spl_handler_SplHeap.dtor_obj = zend_objects_destroy_object;
1210:     spl_handler_SplHeap.free_obj = spl_heap_object_free_storage;
1211:
1212:     REGISTER_SPL_IMPLMENTS(SplHeap, Iterator);
1213:     REGISTER_SPL_IMPLMENTS(SplHeap, Countable);
1214:
1215:     spl_ow_SplHeap->get_iterator = spl_heap_get_iterator;
1216:
1217:     REGISTER_SPL_SUB_CLASS_EX(SplMinHeap, SplHeap, spl_heap_object_new, spl_funcs_SplMinHeap);
1218:     REGISTER_SPL_SUB_CLASS_EX(SplMaxHeap, SplHeap, spl_heap_object_new, spl_funcs_SplMaxHeap);
1219:
1220:     spl_ow_SplMaxHeap->get_iterator = spl_heap_get_iterator;
1221:     spl_ow_SplMinHeap->get_iterator = spl_heap_get_iterator;
1222:
1223:     REGISTER_SPL_STD_CLASS_EX(SplPriorityQueue, spl_heap_object_new, spl_funcs_SplPriorityQueue);
1224:     memcpy(&spl_handler_SplPriorityQueue, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1225:
1226:     spl_handler_SplPriorityQueue.offset = XOffsetOf(spl_heap_object, std);
1227:     spl_handler_SplPriorityQueue.clone_obj = spl_heap_object_clone;
1228:     spl_handler_SplPriorityQueue.count_elements = spl_heap_object_count_elements;
1229:     spl_handler_SplPriorityQueue.get_debug_info = spl_pqqueue_object_get_debug_info;
1230:     spl_handler_SplPriorityQueue.get_gc = spl_heap_object_get_gc;
1231:     spl_handler_SplPriorityQueue.dtor_obj = zend_object_destroy_obj;
1232:     spl_handler_SplPriorityQueue.free_obj = spl_heap_object_free_storage;
1233:
1234:     REGISTER_SPL_IMPLMENTS(SplPriorityQueue, Iterator);
1235:     REGISTER_SPL_IMPLMENTS(SplPriorityQueue, Countable);
1236:
1237:     spl_ow_SplPriorityQueue->get_iterator = spl_pqqueue_get_iterator;
1238:
1239:     REGISTER_SPL_CLASS_CONST_LONG(SplPriorityQueue, "EXTR_BOTH", SPL_PQEXTR_EXTR_BOTH);
1240:     REGISTER_SPL_CLASS_CONST_LONG(SplPriorityQueue, "EXTR_PRIORITY", SPL_PQEXTR_EXTR_PRIORITY);
1241:     REGISTER_SPL_CLASS_CONST_LONG(SplPriorityQueue, "EXTR_DATA", SPL_PQEXTR_EXTR_DATA);
1242:
1243:     return SUCCESS;
1244: }
1245: /* }}} */
1246:
1247: /*
1248:  * Local variables:
1249:  * tab-width: 4
1250:  * c-basic-offset: 4
1251:  * End:
1252:  * vim600: fdm=marker
1253:  * vim: noet sw=4 ts=4
1254:  */
1255: }
```

```

1: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: 65: 66: 67: 68: 69: 70: 71: 72: 73: 74: 75: 76: 77: 78: 79: 80: 81: 82: 83: 84: 85: 86: 87: 88: 89: 90: 91: 92: 93: 94: 95: 96: 97: 98: 99: 100: 101: 102: 103: 104: 105: 106: 107: 108: 109: 110: 111: 112: 113: 114: 115: 116: 117: 118: 119: 120: 121: 122: 123: 124: 125: 126: 127: 128: 129: 130: 131: 132: 133: 134: 135: 136: 137: 138: 139: 140: 141: 142: 143: 144: 145: 146: 147: 148: 149: 150: 151: 152: 153: 154: 155: 156: 157: 158: 159: 160: 161: 162: 163: 164: 165: 166: 167: 168: 169: 170: 171: 172: 173: 174: 175: 176: 177: 178: 179: 180: 181: 182: 183: 184: 185: 186: 187: 188: 189: 190: 191: 192: 193: 194: 195: 196: 197: 198: 199: 200: 201: 202: 203: 204: 205: 206: 207: 208: 209: 210: 211: 212: 213: 214: 215: 216: 217: 218: 219: 220: 221: 222: 223: 224: 225: 226: 227: 228: 229: 230: 231: 232: 233: 234: 235: 236: 237: 238: 239: 240: 241: 242: 243: 244: 245: 246: 247: 248: 249: 250: 251: 252: 253: 254: 255: 256: 257: 258: 259: 260: 261: 262: 263: 264: 265: 266: 267: 268: 269: 270: 271: 272: 273: 274: 275: 276: 277: 278: 279: 280: 281: 282: 283: 284: 285: 286: 287: 288: 289: 290: 291: 292: 293: 294: 295: 296: 297: 298: 299: 300: 301: 302: 303: 304: 305: 306: 307: 308: 309: 310: 311: 312: 313: 314: 315: 316: 317: 318: 319: 320: 321: 322: 323: 324: 325: 326: 327: 328: 329: 330: 331: 332: 333: 334: 335: 336: 337: 338: 339: 340: 341: 342: 343: 344: 345: 346: 347: 348: 349: 350: 351: 352: 353: 354: 355: 356: 357: 358: 359: 360: 361: 362: 363: 364: 365: 366: 367: 368: 369: 370: 371: 372: 373: 374: 375: 376: 377: 378: 379: 380: 381: 382: 383: 384: 385: 386: 387: 388: 389: 390: 391: 392: 393: 394: 395: 396: 397: 398: 399: 400: 401: 402: 403: 404: 405: 406: 407: 408: 409: 410: 411: 412: 413: 414: 415: 416: 417: 418: 419: 420: 421: 422: 423: 424: 425: 426: 427: 428: 429: 430: 431: 432: 433: 434: 435: 436: 437: 438: 439: 440: 441: 442: 443: 444: 445: 446: 447: 448: 449: 450: 451: 452: 453: 454: 455: 456: 457: 458: 459: 460: 461: 462: 463: 464: 465: 466: 467: 468: 469: 470: 471: 472: 473: 474: 475: 476: 477: 478: 479: 480: 481: 482: 483: 484: 485: 486: 487: 488: 489: 490: 491: 492: 493: 494: 495: 496: 497: 498: 499: 500: 501: 502: 503: 504: 505: 506: 507: 508: 509: 510: 511: 512: 513: 514: 515: 516: 517: 518: 519: 520: 521: 522: 523: 524: 525: 526: 527: 528: 529: 530: 531: 532: 533: 534: 535: 536: 537: 538: 539: 540: 541: 542: 543: 544: 545: 546: 547: 548: 549: 550: 551: 552: 553: 554: 555: 556: 557: 558: 559: 560: 561: 562: 563: 564: 565: 566: 567: 568: 569: 570: 571: 572: 573: 574: 575: 576: 577: 578: 579: 580: 581: 582: 583: 584: 585: 586: 587: 588: 589: 590: 591: 592: 593: 594: 595: 596: 597: 598: 599: 600: 601: 602: 603: 604: 605: 606: 607: 608: 609: 610: 611: 612: 613: 614: 615: 616: 617: 618: 619: 620: 621: 622: 623: 624: 625: 626: 627: 628: 629: 630: 631: 632: 633: 634: 635: 636: 637: 638: 639: 640: 641: 642: 643: 644: 645: 646: 647: 648: 649: 650: 651: 652: 653: 654: 655: 656: 657: 658: 659: 660: 661: 662: 663: 664: 665: 666: 667: 668: 669: 670: 671: 672: 673: 674: 675: 676: 677: 678: 679: 680: 681: 682: 683: 684: 685: 686: 687: 688: 689: 690: 691: 692: 693: 694: 695: 696: 697: 698: 699: 700: 701: 702: 703: 704: 705: 706: 707: 708: 709: 710: 711: 712: 713: 714: 715: 716: 717: 718: 719: 720: 721: 722: 723: 724: 725: 726: 727: 728: 729: 730: 731: 732: 733: 734: 735: 736: 737: 738: 739: 740: 741: 742: 743: 744: 745: 746: 747: 748: 749: 750: 751: 752: 753: 754: 755: 756: 757: 758: 759: 760: 761: 762: 763: 764: 765: 766: 767: 768: 769: 770: 771: 772: 773: 774: 775: 776: 777: 778: 779: 780: 781: 782: 783: 784: 785: 786: 787: 788: 789: 790: 791: 792: 793: 794: 795: 796: 797: 798: 799: 800: 801: 802: 803: 804: 805: 806: 807: 808: 809: 810: 811: 812: 813: 814: 815: 816: 817: 818: 819: 820: 821: 822: 823: 824: 825: 826: 827: 828: 829: 830: 831: 832: 833: 834: 835: 836: 837: 838: 839: 840: 841: 842: 843: 844: 845: 846: 847: 848: 849: 850: 851: 852: 853: 854:
```

```

377: return spl_object_storage_new_ex(class_type, NULL);
378: }
379: /* }}} */
380:
381: int spl_object_storage_contains(spl_SplObjectStorage *intern, zval *this, zval *obj) /* {{{ */
382: {
383:     int found;
384:     zend_hash_key_key;
385:     if (spl_object_storage_get_hash(key, intern, this, obj) == FAILURE) {
386:         return 0;
387:     }
388:     if (key.key) {
389:         found = zend_hash_exists(intern->storage, key.key);
390:     } else {
391:         found = zend_hash_index_exists(intern->storage, key.h);
392:     }
393:     if (found) {
394:         spl_object_storage_free_hash(intern, &key);
395:         return found;
396:     } /* }}} */
397:
398: /* {{{ proto void SplObjectStorage::attach(object obj, mixed inf = NULL)
399:  Attaches an object to the storage if not yet contained */
400: SPL_METHOD(spl_object_storage_attach)
401: {
402:     zval *obj, *inf = NULL;
403:
404:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
405:
406:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o!i", &obj, &inf) == FAILURE) {
407:         return;
408:     }
409:     spl_object_storage_attach(intern, getThis(), obj, inf);
410: } /* }}} */
411:
412: /* {{{ proto void SplObjectStorage::detach(object obj)
413:  Detaches an object from the storage */
414: SPL_METHOD(spl_object_storage_detach)
415: {
416:     zval *obj;
417:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
418:
419:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
420:         return;
421:     }
422:     spl_object_storage_detach(intern, getThis(), obj);
423:
424:     zend_hash_internal_pointer_reset_ex(intern->storage, &intern->pos);
425:     intern->index = 0;
426: } /* }}} */
427:
428: /* {{{ proto string SplObjectStorage::getHash(object obj)
429:  Returns the hash of an object */
430: SPL_METHOD(spl_object_storage_getHash)
431: {
432:     zval *obj;
433:
434:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
435:         return;
436:     }
437:
438:     RETURN_NEW_STR(PHP_SPL_OBJECT_HASH(obj));
439: } /* }}} */
440:
441: /* {{{ proto mixed SplObjectStorage::offsetGet(object obj)
442:  Returns associated information for a stored object */
443: SPL_METHOD(spl_object_storage_offsetGet)
444: {
445:     zval *obj;
446:     spl_SplObjectStorageElement *element;
447:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
448:     zend_hash_key_key;
449:
450:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
451:         return;
452:     }
453:
454:     if (spl_object_storage_get_hash(key, intern, getThis(), obj) == FAILURE) {
455:         return;
456:     }
457:
458:     element = spl_object_storage_get(intern, &key);
459:     spl_object_storage_free_hash(intern, &key);
460:
461:     if (!element) {
462:         zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0, "Object not found");
463:     } else {
464:         zval *value = element->inf;
465:
466:         ZVAL_DEREF(value);
467:         ZVAL_COPY(&return_value, value);
468:     }
469: } /* }}} */
470:
471: /* {{{ proto bool SplObjectStorage::addAll(SplObjectStorage $os)
472:  Add all elements contained in $os */
473: SPL_METHOD(spl_object_storage_addAll)
474: {
475:     zval *obj;
476:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
477:     spl_SplObjectStorage *other;
478:     spl_SplObjectStorageElement *element;
479:
480:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, &spl_ce_SplObjectStorage) == FAILURE) {
481:         return;
482:     }
483:
484:     other = Z_SPLOBJSTORAGE_P(obj);
485:
486:     spl_object_storage_addAll(intern, getThis(), other);
487:
488:     RETURN_LONG(zend_hash_num_elements(intern->storage));
489: } /* }}} */
490:
491: /* {{{ proto bool SplObjectStorage::removeAll(SplObjectStorage $os)
492:  Remove all elements contained in $os */
493: SPL_METHOD(spl_object_storage_removeAll)
494: {
495:     zval *obj;
496:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
497:     spl_SplObjectStorage *other;
498:     spl_SplObjectStorageElement *element;
499:
500:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, &spl_ce_SplObjectStorage) == FAILURE) {
501:         return;
502:     }
503:
504:     other = Z_SPLOBJSTORAGE_P(obj);
505:
506:     zend_hash_internal_pointer_reset_ex(other->storage);
507:     while ((element = zend_hash_get_current_data_ptr_ex(other->storage)) != NULL) {
508:         if (spl_object_storage_detach(intern, getThis(), element->obj) == FAILURE) {
509:             zend_hash_move_forward(other->storage);
510:         }
511:     }
512:
513:     zend_hash_internal_pointer_reset_ex(intern->storage, &intern->pos);
514:     intern->index = 0;
515:
516:     RETURN_LONG(zend_hash_num_elements(intern->storage));
517: } /* }}} */
518:
519: /* {{{ proto bool SplObjectStorage::removeAllExcept(SplObjectStorage $os)
520:  Remove elements not common to both this SplObjectStorage instance and $os */
521: SPL_METHOD(spl_object_storage_removeAllExcept)
522: {
523:     zval *obj;
524:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
525:     spl_SplObjectStorage *other;
526:     spl_SplObjectStorageElement *element;
527:
528:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, &spl_ce_SplObjectStorage) == FAILURE) {
529:         return;
530:     }
531:
532:     other = Z_SPLOBJSTORAGE_P(obj);
533:
534:     ZEND_HASH_FOREACH_PTR(intern->storage, element) {
535:         if (!spl_object_storage_contains(other, getThis(), element->obj)) {
536:             spl_object_storage_detach(intern, getThis(), element->obj);
537:         }
538:     } ZEND_HASH_FOREACH_END();
539:
540:     zend_hash_internal_pointer_reset_ex(intern->storage, &intern->pos);
541:     intern->index = 0;
542:
543:     RETURN_LONG(zend_hash_num_elements(intern->storage));
544: } /* }}} */
545:
546: /* {{{ proto bool SplObjectStorage::contains(object obj)
547:  Determine whether an object is contained in the storage */
548: SPL_METHOD(spl_object_storage_contains)
549: {
550:     zval *obj;
551:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
552:
553:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
554:         return;
555:     }
556:
557:     RETURN_BOOL(spl_object_storage_contains(intern, getThis(), obj));
558: } /* }}} */
559:
560: /* {{{ proto int SplObjectStorage::count()
561:  Determine number of objects in storage */
562: SPL_METHOD(spl_object_storage_count)
563: {
564:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
565:
566:     zend_long mode = COUNT_NORMAL;
567:
568:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "!i", &mode) == FAILURE) {
569:         return;
570:     }
571:
572:     if (mode == COUNT_RECURSIVE) {
573:         zend_long ret;
574:
575:         if (mode != COUNT_RECURSIVE) {
576:             ret = zend_hash_num_elements(intern->storage);
577:         } else {
578:             ret = php_count_recursive(intern->storage);
579:         }
580:
581:         RETURN_LONG(ret);
582:     }
583:
584:     RETURN_LONG(zend_hash_num_elements(intern->storage));
585: } /* }}} */
586:
587: /* {{{ proto void SplObjectStorage::rewind()
588:  Rewind to first position */
589: SPL_METHOD(spl_object_storage_rewind)
590: {
591:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
592:
593:     if (zend_parse_parameters_none() == FAILURE) {
594:         return;
595:     }
596:
597:     zend_hash_internal_pointer_reset_ex(intern->storage, &intern->pos);
598:     intern->index = 0;
599: } /* }}} */
600:
601: /* {{{ proto bool SplObjectStorage::valid()
602:  Returns whether current position is valid */
603: SPL_METHOD(spl_object_storage_valid)
604: {
605:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
606:
607:     if (zend_parse_parameters_none() == FAILURE) {
608:         return;
609:     }
610:
611:     RETURN_BOOL(zend_hash_has_more_elements_ex(intern->storage, &intern->pos) == SUCCESS);
612: } /* }}} */
613:
614: /* {{{ proto mixed SplObjectStorage::key()
615:  Returns current key */
616: SPL_METHOD(spl_object_storage_key)
617: {
618:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
619:
620:     if (zend_parse_parameters_none() == FAILURE) {
621:         return;
622:     }
623:
624:     RETURN_LONG(intern->index);
625: } /* }}} */
626:
627: /* {{{ proto mixed SplObjectStorage::current()
628:  Returns current element */
629: SPL_METHOD(spl_object_storage_current)
630: {
631:     spl_SplObjectStorageElement *element;
632:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
633:
634:     if (zend_parse_parameters_none() == FAILURE) {
635:         return;
636:     }
637:
638:     if ((element = zend_hash_get_current_data_ptr_ex(intern->storage, &intern->pos)) == NULL) {
639:         return;
640:     }
641:     ZVAL_COPY(&return_value, element->obj);
642: } /* }}} */
643:
644: /* {{{ proto mixed SplObjectStorage::getInfo()
645:  Returns associated information to current element */
646: SPL_METHOD(spl_object_storage_getInfo)
647: {
648:     spl_SplObjectStorageElement *element;
649:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
650:
651:     if (zend_parse_parameters_none() == FAILURE) {
652:         return;
653:     }
654:
655:     if ((element = zend_hash_get_current_data_ptr_ex(intern->storage, &intern->pos)) == NULL) {
656:         return;
657:     }
658:     ZVAL_COPY(&return_value, element->inf);
659: } /* }}} */
660:
661: /* {{{ proto mixed SplObjectStorage::setInfo(mixed $inf)
662:  Sets associated information of current element to $inf */
663: SPL_METHOD(spl_object_storage_setInfo)
664: {
665:     spl_SplObjectStorageElement *element;
666:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
667:     zval *inf;
668:
669:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &inf) == FAILURE) {
670:         return;
671:     }
672:
673:     if ((element = zend_hash_get_current_data_ptr_ex(intern->storage, &intern->pos)) == NULL) {
674:         return;
675:     }
676:     zval *ptr = element->inf;
677:     ZVAL_COPY(&element->inf, inf);
678: } /* }}} */
679:
680: /* {{{ proto void SplObjectStorage::next()
681:  Moves position forward */
682: SPL_METHOD(spl_object_storage_next)
683: {
684:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
685:
686:     if (zend_parse_parameters_none() == FAILURE) {
687:         return;
688:     }
689:
690:     zend_hash_move_forward_ex(intern->storage, &intern->pos);
691:     intern->index++;
692: } /* }}} */
693:
694: /* {{{ proto string SplObjectStorage::serialize()
695:  Serializes storage */
696: SPL_METHOD(spl_object_storage_serialize)
697: {
698:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
699:
700:     spl_SplObjectStorageElement *element;
701:     zval members, flags;
702:     HashPosition pos;
703:     PHP_SERIALIZE_DATA var_hash;
704:     smart_str buf = 0;
705:
706:     if (zend_parse_parameters_none() == FAILURE) {
707:         return;
708:     }
709:
710:     PHP_VAR_SERIALIZE_INIT(var_hash);
711:
712:     /* storage */
713:     smart_str_append(&buf, "s:", 2);
714:     ZVAL_LONG(&flags, zend_hash_num_elements(intern->storage));
715:     PHP_VAR_SERIALIZE(&buf, &flags, &var_hash);
716:     zval_ptr_stor(&flags);
717:
718:     zend_hash_internal_pointer_reset_ex(intern->storage, &pos);
719:
720:     while (zend_hash_has_more_elements_ex(intern->storage, &pos) == SUCCESS) {
721:         if ((element = zend_hash_get_current_data_ptr_ex(intern->storage, &pos)) == NULL) {
722:             smart_str_free(&buf);
723:             PHP_VAR_SERIALIZE_DESTROY(var_hash);
724:             RETURN_NULL();
725:         }
726:         php_var_serialize(&buf, element->obj, &var_hash);
727:         smart_str_append(&buf, ":", 1);
728:         php_var_serialize(&buf, element->inf, &var_hash);
729:         smart_str_append(&buf, ":", 1);
730:         zend_hash_move_forward_ex(intern->storage, &pos);
731:     }
732:
733:     /* members */
734:     smart_str_append(&buf, "m:", 2);
735:
736:     ZVAL_ARR(&members, zend_array_dup(zend_std_get_grogeties(getThis())));
737:     php_var_serialize(&buf, &members, &var_hash); /* finishes the string */
738:     zval_ptr_stor(&members);
739:
740:     /* done */
741:     PHP_VAR_SERIALIZE_DESTROY(var_hash);
742:
743:     if (&buf.s) {
744:         RETURN_NEW_STR(&buf.s);
745:     } else {
746:         RETURN_NULL();
747:     }
748: } /* }}} */
749:
750: /* {{{ proto void SplObjectStorage::unserialize(string $serialized)
751:  Unserializes storage */

```

```

932: typedef enum {
933:     MIT_NEED_ANY      = 0,
934:     MIT_NEED_ALL      = 1,
935:     MIT_KEYS_NUMERIC  = 0,
936:     MIT_KEYS_ASSOC    = 2,
937: } MultipleIteratorFlags;
938:
939: #define SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT 1
940: #define SPL_MULTIPLE_ITERATOR_GET_ALL_KEY

```

```

1120: array_init_size(return_value, num_elements);
1121:
1122: zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1123: while (element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) != NULL && !EG(exception) {
1124:     it = element->obj;
1125:     zend_call_method_with_0_params(it, Z_OBJCE_P(it), z OBJC_P(it)->iterator_funcs.zf_valid, "valid", &retval);
1126:
1127:     if (!IS_TRUE(retval))
1128:         break;
1129: }

```



```

1129:     rval_get_dtor(&retval);
1130: } else {
1131:     valid = 0;
1132: }
1133:
1134: if (valid) {
1135:     if (SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT == get_type) {
1136:         send_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs.if_current, "current", &retval);
1137:     } else {
1138:         send_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs.if_key, "key", &retval);
1139:     }
1140:     if (!Z_ISUNDEF(&retval)) {
1141:         zend_throw_exception(spl_ce_RuntimeException, "Failed to call sub iterator method", 0);
1142:         return;
1143:     }
1144: } else if (intern->flags & MIT_NEED_ALL) {
1145:     if (SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT == get_type) {
1146:         zend_throw_exception(spl_ce_RuntimeException, "Called current() with non valid sub iterator", 0);
1147:     } else {
1148:         zend_throw_exception(spl_ce_RuntimeException, "Called key() with non valid sub iterator", 0);
1149:     }
1150:     return;
1151: } else {
1152:     ZVAL_NULL(&retval);
1153: }
1154:
1155: if (intern->flags & MIT_KEYS_ASSOC) {
1156:     switch (Z_TYPE(element->info)) {
1157:         case IS_LONG:
1158:             add_index_zval(return_value, Z_LVAL(element->info), &retval);
1159:             break;
1160:         case IS_STRING:
1161:             zend_symtable_update(&ARRVAL_P(return_value), Z_STR(element->info), &retval);
1162:             break;
1163:         default:
1164:             rval_get_dtor(&retval);
1165:             zend_throw_exception(spl_ce_InvalidArgumentException, "Sub-Iterator is associated with NULL", 0);
1166:             return;
1167:     }
1168: } else {
1169:     add_next_index_zval(return_value, &retval);
1170: }
1171:
1172: zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1173: }
1174: /* }}} */
1175: /* }}} */
1176:
1177: /* {{{ proto array current() throws RuntimeException throws InvalidArgumentException
1178:    Return an array of all registered Iterator instances current() result */
1179: SPL_METHOD(MultipleIterator, current)
1180: {
1181:     spl_SplObjectStorage *intern;
1182:     intern = Z_SPL_OBJECT_STORAGE_P(getThis());
1183:
1184:     if (zend_parse_parameters_none() == FAILURE) {
1185:         return;
1186:     }
1187:
1188:     spl_multiple_iterator_get_all(intern, SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT, return_value);
1189: }
1190: /* }}} */
1191:
1192: /* {{{ proto array MultipleIterator::key()
1193:    Return an array of all registered Iterator instances key() result */
1194: SPL_METHOD(MultipleIterator, key)
1195: {
1196:     spl_SplObjectStorage *intern;
1197:     intern = Z_SPL_OBJECT_STORAGE_P(getThis());
1198:
1199:     if (zend_parse_parameters_none() == FAILURE) {
1200:         return;
1201:     }
1202:
1203:     spl_multiple_iterator_get_all(intern, SPL_MULTIPLE_ITERATOR_GET_ALL_KEY, return_value);
1204: }
1205: /* }}} */
1206:
1207: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_attachIterator, 0, 0, 1)
1208:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
1209: ZEND_END_ARG_INFO();
1210: ZEND_FUNC_INFO();
1211:
1212: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_detachIterator, 0, 0, 1)
1213:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
1214: ZEND_END_ARG_INFO();
1215:
1216: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_containsIterator, 0, 0, 1)
1217:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
1218: ZEND_END_ARG_INFO();
1219:
1220: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_setFlags, 0, 0, 1)
1221:     ZEND_ARG_INFO(0, flags)
1222: ZEND_END_ARG_INFO();
1223:
1224: static const zend_function_entry spl_funcs_MultipleIterator[] = {
1225:     SPL_ME(MultipleIterator, __construct,      arginfo_MultipleIterator_setFlags,      0)
1226:     SPL_ME(MultipleIterator, getFlags,         arginfo_splObject_void,                0)
1227:     SPL_ME(MultipleIterator, setFlags,         arginfo_MultipleIterator_setFlags,     0)
1228:     SPL_ME(MultipleIterator, attachIterator,   arginfo_MultipleIterator_attachIterator, 0)
1229:     SPL_ME(MultipleIterator, detachIterator,   splObjectStorage, detach,      arginfo_MultipleIterator_detachIterator, 0)
1230:     SPL_ME(MultipleIterator, containsIterator, splObjectStorage, contains,   arginfo_MultipleIterator_containsIterator, 0)
1231:     SPL_ME(MultipleIterator, countIterators,  splObjectStorage, count,      arginfo_splObject_void,            0)
1232:     /* Iterator */
1233:     SPL_ME(MultipleIterator, rewind,           arginfo_splObject_void,                0)
1234:     SPL_ME(MultipleIterator, valid,           arginfo_splObject_void,                0)
1235:     SPL_ME(MultipleIterator, key,             arginfo_splObject_void,                0)
1236:     SPL_ME(MultipleIterator, current,         arginfo_splObject_void,                0)
1237:     SPL_ME(MultipleIterator, next,            arginfo_splObject_void,                0)
1238:     PHP_FE_END
1239: };
1240:
1241: /* {{{ PHP_MINIT_FUNCTION(spl_observer) */
1242: PHP_MINIT_FUNCTION(spl_observer)
1243: {
1244:     REGISTER_SPL_INTERFACE(splObserver);
1245:     REGISTER_SPL_INTERFACE(splSubject);
1246:
1247:     REGISTER_SPL_STD_CLASS_EX(splObjectStorage, spl_SplObjectStorage_new, spl_funcs_splObjectStorage);
1248:     memcpy(&spl_handler_splObjectStorage, &zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1249:
1250:     spl_handler_splObjectStorage.offset = XOffsetOf(spl_SplObjectStorage, std);
1251:     spl_handler_splObjectStorage.get_debug_info = spl_object_storage_debug_info;
1252:     spl_handler_splObjectStorage.compare_objects = spl_object_storage_compare_objects;
1253:     spl_handler_splObjectStorage.clone_obj = spl_object_storage_clone;
1254:     spl_handler_splObjectStorage.get_gc = spl_object_storage_get_gc;
1255:     spl_handler_splObjectStorage.dtor_obj = zend_object_dtor_obj;
1256:     spl_handler_splObjectStorage.free_obj = spl_SplObjectStorage_free_storage;
1257:
1258:     REGISTER_SPL_IMPLEMENTATIONS(splObjectStorage, Countable);
1259:     REGISTER_SPL_IMPLEMENTATIONS(splObjectStorage, Iterator);
1260:     REGISTER_SPL_IMPLEMENTATIONS(splObjectStorage, Serializable);
1261:     REGISTER_SPL_IMPLEMENTATIONS(splObjectStorage, ArrayAccess);
1262:
1263:     REGISTER_SPL_STD_CLASS_EX(MultipleIterator, spl_SplObjectStorage_new, spl_funcs_MultipleIterator);
1264:     REGISTER_SPL_ITERATOR(MultipleIterator);
1265:
1266:     REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_NEED_ANY", MIT_NEED_ANY);
1267:     REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_NEED_ALL", MIT_NEED_ALL);
1268:     REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_KEYS_NUMERIC", MIT_KEYS_NUMERIC);
1269:     REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_KEYS_ASSOC", MIT_KEYS_ASSOC);
1270:
1271:     return SUCCESS;
1272: }
1273: /* }}} */
1274:
1275: /*
1276:  * Local variables:
1277:  * tab-width: 4
1278:  * c-basic-offset: 4
1279:  * End:
1280:  * vim600: fdm=marker
1281:  * vim: noet sw=4 ts=4
1282:  */

```

```

1: 190: }
2: 191: return current;
3: 192: }
4: 193: /* }}} */
5: 194:
6: 195: static void spl_ptr_list_unshift(spl_ptr_list *llist, zval *data) /* {{{ */
7: 196: {
8: 197:     spl_ptr_list_element *elem = emalloc(sizeof(spl_ptr_list_element));
9: 198:
10: 199:     elem->rc = 1;
11: 200:     elem->prev = NULL;
12: 201:     elem->next = llist->head;
13: 202:     ZVAL_COPY_VALUE(elem->data, data);
14: 203:
15: 204:     if (llist->head) {
16: 205:         llist->head->prev = elem;
17: 206:     } else {
18: 207:         llist->tail = elem;
19: 208:     }
20: 209:
21: 210:     llist->head = elem;
22: 211:     llist->count++;
23: 212:
24: 213:     if (llist->vctor) {
25: 214:         llist->vctor(elem);
26: 215:     }
27: 216: }
28: 217: /* }}} */
29: 218:
30: 219: static void spl_ptr_list_push(spl_ptr_list *llist, zval *data) /* {{{ */
31: 220: {
32: 221:     spl_ptr_list_element *elem = emalloc(sizeof(spl_ptr_list_element));
33: 222:
34: 223:     elem->rc = 1;
35: 224:     elem->prev = llist->tail;
36: 225:     elem->next = NULL;
37: 226:     ZVAL_COPY_VALUE(elem->data, data);
38: 227:
39: 228:     if (llist->tail) {
40: 229:         llist->tail->next = elem;
41: 230:     } else {
42: 231:         llist->head = elem;
43: 232:     }
44: 233:
45: 234:     llist->tail = elem;
46: 235:     llist->count++;
47: 236:
48: 237:     if (llist->vctor) {
49: 238:         llist->vctor(elem);
50: 239:     }
51: 240: }
52: 241: /* }}} */
53: 242:
54: 243: static void spl_ptr_list_pop(spl_ptr_list *llist, zval *ret) /* {{{ */
55: 244: {
56: 245:     spl_ptr_list_element *tail = llist->tail;
57: 246:
58: 247:     if (tail == NULL) {
59: 248:         return;
60: 249:     }
61: 250:
62: 251:     if (tail->prev) {
63: 252:         tail->prev->next = NULL;
64: 253:     } else {
65: 254:         llist->head = NULL;
66: 255:     }
67: 256:
68: 257:     llist->tail = tail->prev;
69: 258:     llist->count--;
70: 259:     ZVAL_COPY(ret, tail->data);
71: 260:
72: 261:     if (llist->vctor) {
73: 262:         llist->vctor(tail);
74: 263:     }
75: 264:
76: 265:     ZVAL_UNREF(tail->data);
77: 266:
78: 267:     SPL_LIST_DELREF(tail);
79: 268:
80: 269:     return;
81: 270: }
82: 271: /* }}} */
83: 272: static zval *spl_ptr_list_last(spl_ptr_list *llist) /* {{{ */
84: 273: {
85: 274:     spl_ptr_list_element *tail = llist->tail;
86: 275:
87: 276:     if (tail == NULL) {
88: 277:         return NULL;
89: 278:     } else {
90: 279:         return tail->data;
91: 280:     }
92: 281: }
93: 282: /* }}} */
94: 283:
95: 284: static zval *spl_ptr_list_first(spl_ptr_list *llist) /* {{{ */
96: 285: {
97: 286:     spl_ptr_list_element *head = llist->head;
98: 287:
99: 288:     if (head == NULL) {
100: 289:         return NULL;
101: 290:     } else {
102: 291:         return head->data;
103: 292:     }
104: 293: }
105: 294: /* }}} */
106: 295:
107: 296: static void spl_ptr_list_shift(spl_ptr_list *llist, zval *ret) /* {{{ */
108: 297: {
109: 298:     spl_ptr_list_element *head = llist->head;
109: 299:
110: 300:     if (head == NULL) {
111: 301:         return;
112: 302:     }
113: 303:
114: 304:     if (head->next) {
115: 305:         head->next->prev = NULL;
116: 306:     } else {
117: 307:         llist->tail = NULL;
118: 308:     }
119: 309:
120: 310:     llist->head = head->next;
121: 311:     llist->count--;
122: 312:     ZVAL_COPY(ret, head->data);
123: 313:
124: 314:     if (llist->vctor) {
125: 315:         llist->vctor(head);
126: 316:     }
127: 317:
128: 318:     ZVAL_UNREF(head->data);
129: 319:
130: 320:     SPL_LIST_DELREF(head);
131: 321: }
132: 322: /* }}} */
133: 323:
134: 324: static void spl_ptr_list_copy(spl_ptr_list *from, spl_ptr_list *to) /* {{{ */
135: 325: {
136: 326:     spl_ptr_list_element *current = from->head, *next;
137: 327:     /*??? spl_ptr_list_ctor_func ctor = from->ctor;
138: 328:
139: 329:     while (current) {
140: 330:         next = current->next;
141: 331:
142: 332:         /*??? FIXME
143: 333:         if (ctor) {
144: 334:             ctor(current);
145: 335:         }
146: 336:         */
147: 337:
148: 338:         spl_ptr_list_push(to, current->data);
149: 339:         current = next;
150: 340:     }
151: 341:
152: 342:     if (to) {
153: 343:         /* }}} */
154: 344:
155: 345:         /* }}} */
156: 346:
157: 347: static void spl_dlist_object_free_storage(zend_object *obj) /* {{{ */
158: 348: {
159: 349:     spl_dlist_object *intern = spl_dlist_from_obj(obj);
160: 350:     zval tmp;
161: 351:
162: 352:     zend_object_std_dtor(intern->std);
163: 353:
164: 354:     while (intern->llist->count > 0) {
165: 355:         spl_ptr_list_pop(intern->llist, &tmp);
166: 356:         zval_ptr_dtor(&tmp);
167: 357:     }
168: 358:
169: 359:     if (intern->gc_data != NULL) {
170: 360:         efree(intern->gc_data);
171: 361:     }
172: 362:
173: 363:     spl_ptr_list_destroy(intern->llist);
174: 364:     SPL_LIST_CHECK_DELREF(intern->traverse_pointer);
175: 365: }
176: 366: /* }}} */
177: 367:
178: 368: zend_object_iterator *spl_dlist_get_iterator(zend_class_entry *ce, zval *obj,
179: 369:
180: 370: static zend_object *spl_dlist_object_new_ex(zend_class_entry *class_type, zval
181: 371: {
182: 372:     spl_dlist_object *intern;
183: 373:     zend_class_entry *parent = class_type;
184: 374:     int
185: 375:         inherited = 0;
186: 376:
187: 377:     intern = zend_object_alloc(sizeof(spl_dlist_object), parent);
188: 378:

```

```

377: zend_object_std_init(&intern->std, class_type);
378: object_properties_init(&intern->std, class_type);
379:
380:
381: intern->flags = 0;
382: intern->traverse_position = 0;
383:
384: if (orig) {
385:     spl_dlist_object *other = Z_SPDOLLIST_P(orig);
386:     intern->ow_get_iterator = other->ow_get_iterator;
387:
388:     if (clone_orig) {
389:         intern->llist = (spl_ptr_llist *)spl_ptr_llist_init(&other->llist->ctor, &other->llist->dtor);
390:         spl_ptr_llist_copy(&other->llist, intern->llist);
391:         intern->traverse_pointer = intern->llist->thead;
392:         SPL_LLIST_CHECK_ADDRP(intern->traverse_pointer);
393:     } else {
394:         intern->llist = other->llist;
395:         intern->traverse_pointer = intern->llist->thead;
396:         SPL_LLIST_CHECK_ADDRP(intern->traverse_pointer);
397:     }
398:
399:     intern->flags = other->flags;
400: } else {
401:     intern->llist = (spl_ptr_llist *)spl_ptr_llist_init(spl_ptr_llist_ewl_ctor, spl_ptr_llist_ewl_dtor);
402:     intern->traverse_pointer = intern->llist->thead;
403:     SPL_LLIST_CHECK_ADDRP(intern->traverse_pointer);
404: }
405:
406: while (parent) {
407:     if (parent == spl_ow_SplStack) {
408:         intern->flags |= (SPL_DLIST_IT_FIX | SPL_DLIST_IT_IFPO);
409:         intern->std.handlers = spl_handler_SplDoublyLinkedList;
410:     } else if (parent == spl_ow_SplQueue) {
411:         intern->flags |= (SPL_DLIST_IT_FIX);
412:         intern->std.handlers = spl_handler_SplDoublyLinkedList;
413:     }
414:
415:     if (parent == spl_ow_SplDoublyLinkedList) {
416:         intern->std.handlers = spl_handler_SplDoublyLinkedList;
417:         break;
418:     }
419:
420:     parent = parent->parent;
421:     inherited = 1;
422: }
423:
424: if (!parent) { /* this must never happen */
425:     php_error_docref(NULL, E_COMPILE_ERROR, "Internal compiler error. Class is not child of SplDoublyLinkedList");
426: }
427:
428: if (inherited) {
429:     intern->fptr_offset_get = zend_hash_str_find_ptr(class_type->function_table, "offsetget", sizeof("offsetget") - 1);
430:     if (intern->fptr_offset_get->common.scope == parent) {
431:         intern->fptr_offset_get = NULL;
432:     }
433:     intern->fptr_offset_set = zend_hash_str_find_ptr(class_type->function_table, "offsetset", sizeof("offsetset") - 1);
434:     if (intern->fptr_offset_set->common.scope == parent) {
435:         intern->fptr_offset_set = NULL;
436:     }
437:     intern->fptr_offset_has = zend_hash_str_find_ptr(class_type->function_table, "offsetexists", sizeof("offsetexists") - 1);
438:     if (intern->fptr_offset_has->common.scope == parent) {
439:         intern->fptr_offset_has = NULL;
440:     }
441:     intern->fptr_offset_del = zend_hash_str_find_ptr(class_type->function_table, "offsetunset", sizeof("offsetunset") - 1);
442:     if (intern->fptr_offset_del->common.scope == parent) {
443:         intern->fptr_offset_del = NULL;
444:     }
445:     intern->fptr_count = zend_hash_str_find_ptr(class_type->function_table, "count", sizeof("count") - 1);
446:     if (intern->fptr_count->common.scope == parent) {
447:         intern->fptr_count = NULL;
448:     }
449: }
450:
451: return &intern->std;
452: } /* }}} */
453:
454: static zend_object *spl_dlist_object_new(zend_class_entry *class_type) /* {{{ */
455: {
456:     return spl_dlist_object_new_ex(class_type, NULL, 0);
457: }
458: /* }}} */
459:
460: static zend_object *spl_dlist_object_clone(zval *zobject) /* {{{ */
461: {
462:     zend_object *old_object;
463:     zend_object *new_object;
464:
465:     old_object = Z_OBJ_P(zobject);
466:     new_object = spl_dlist_object_new_ex(old_object->ce, zobject, 1);
467:
468:     zend_objects_clone_members(new_object, old_object);
469:
470:     return new_object;
471: }
472: /* }}} */
473:
474: static int spl_dlist_object_count_elements(zval *zobject, zend_long *count) /* {{{ */
475: {
476:     spl_dlist_object *intern = Z_SPDOLLIST_P(zobject);
477:
478:     if (intern->fptr_count) {
479:         zval rv;
480:         zend_call_method_with_0_params(object, intern->std.ow, intern->fptr_count, "count", &rv);
481:         if (!Z_ISUNDEF(rv)) {
482:             *count = zval_get_long(&rv);
483:             zval_ptr_dtor(&rv);
484:             return SUCCESS;
485:         }
486:         *count = 0;
487:         return FAILURE;
488:     }
489:
490:     *count = spl_ptr_llist_count(intern->llist);
491:     return SUCCESS;
492: }
493: /* }}} */
494:
495: static HashTable* spl_dlist_object_get_debug_info(zval *obj, int *is_temp) /* {{{ */
496: {
497:     spl_dlist_object *intern = Z_SPDOLLIST_P(obj);
498:     spl_ptr_llist_element *current = intern->llist->thead;
499:     zval tmp, dlist_array;
500:     zend_string *pnstr;
501:     int i = 0;
502:     HashTable *debug_info;
503:     *is_temp = 1;
504:
505:     if (!(&intern->std.properties)) {
506:         rebuild_object_properties(&intern->std);
507:     }
508:
509:     debug_info = zend_new_array(1);
510:     zend_hash_copy(debug_info, &intern->std.properties, (copy_ctor_func_t) zval_add_ref);
511:
512:     pnstr = spl_gen_private_prop_name(spl_ow_SplDoublyLinkedList, "Flags", sizeof("Flags")-1);
513:     ZVAL_LONG(&tmp, intern->flags);
514:     zend_hash_add(debug_info, pnstr, &tmp);
515:     zend_string_release(pnstr);
516:
517:     array_init(&dlist_array);
518:
519:     while (current) {
520:         next = current->next;
521:
522:         add_index_zval(&dlist_array, i, current->data);
523:         if (Z_REFCOUNTED(current->data)) {
524:             Z_ADDREF(current->data);
525:         }
526:         i++;
527:         current = next;
528:     }
529:
530:     pnstr = spl_gen_private_prop_name(spl_ow_SplDoublyLinkedList, "dlist", sizeof("dlist")-1);
531:     zend_hash_add(debug_info, pnstr, &dlist_array);
532:     zend_string_release(pnstr);
533:
534:     return debug_info;
535: }
536: /* }}} */
537:
538: static HashTable *spl_dlist_object_get_gc(zval *obj, zval **gc_data, int *gc_data_count) /* {{{ */
539: {
540:     spl_dlist_object *intern = Z_SPDOLLIST_P(obj);
541:     spl_ptr_llist_element *current = intern->llist->thead;
542:     int i = 0;
543:
544:     if (intern->gc_data_count < intern->llist->count) {
545:         intern->gc_data_count = intern->llist->count;
546:         intern->gc_data = safe_erealloc(intern->gc_data, intern->gc_data_count, sizeof(zval), 0);
547:     }
548:
549:     while (current) {
550:         ZVAL_COPY_VALUE(intern->gc_data[i++], &current->data);
551:         current = current->next;
552:     }
553:
554:     *gc_data = intern->gc_data;
555:     *gc_data_count = i;
556:     return zend_std_get_properties(obj);
557: }
558: /* }}} */
559:
560: /* {{{ proto bool SplDoublyLinkedList::push(mixed value)
561: Push value on the SplDoublyLinkedList */
562: SPL_METHOD(SplDoublyLinkedList, push)
563: {
564:     zval *value;
565:     spl_dlist_object *intern;
566:
567:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &value) == FAILURE) {
568:         return;
569:     }
570:
571:     intern = Z_SPDOLLIST_P(getThis());
572:     spl_ptr_llist_push(intern->llist, value);
573:
574:     RETURN_TRUE;
575: }
576:
577: /* }}} */
578:
579: /* {{{ proto bool SplDoublyLinkedList::unshift(mixed value)
580: Unshift value on the SplDoublyLinkedList */
581: SPL_METHOD(SplDoublyLinkedList, unshift)
582: {
583:     zval *value;
584:     spl_dlist_object *intern;
585:
586:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &value) == FAILURE) {
587:         return;
588:     }
589:
590:     intern = Z_SPDOLLIST_P(getThis());
591:     spl_ptr_llist_unshift(intern->llist, value);
592:
593:     RETURN_TRUE;
594: }
595:
596: /* }}} */
597:
598: /* {{{ proto mixed SplDoublyLinkedList::pop()
599: Pop an element out of the SplDoublyLinkedList */
600: SPL_METHOD(SplDoublyLinkedList, pop)
601: {
602:     spl_dlist_object *intern;
603:
604:     if (zend_parse_parameters_none() == FAILURE) {
605:         return;
606:     }
607:
608:     intern = Z_SPDOLLIST_P(getThis());
609:     spl_ptr_llist_pop(intern->llist, return_value);
610:
611:     if (Z_ISUNDEF_P(return_value)) {
612:         zend_throw_exception(spl_ow_RuntimeException, "Can't pop from an empty datastructure", 0);
613:         RETURN_NULL();
614:     }
615:
616:     /* }}} */
617:
618: /* {{{ proto mixed SplDoublyLinkedList::shift()
619: Shift an element out of the SplDoublyLinkedList */
620: SPL_METHOD(SplDoublyLinkedList, shift)
621: {
622:     spl_dlist_object *intern;
623:
624:     if (zend_parse_parameters_none() == FAILURE) {
625:         return;
626:     }
627:
628:     intern = Z_SPDOLLIST_P(getThis());
629:     spl_ptr_llist_shift(intern->llist, return_value);
630:
631:     if (Z_ISUNDEF_P(return_value)) {
632:         zend_throw_exception(spl_ow_RuntimeException, "Can't shift from an empty datastructure", 0);
633:         RETURN_NULL();
634:     }
635:
636:     /* }}} */
637:
638: /* {{{ proto mixed SplDoublyLinkedList::top()
639: Peek at the top element of the SplDoublyLinkedList */
640: SPL_METHOD(SplDoublyLinkedList, top)
641: {
642:     zval *value;
643:     spl_dlist_object *intern;
644:
645:     if (zend_parse_parameters_none() == FAILURE) {
646:         return;
647:     }
648:
649:     intern = Z_SPDOLLIST_P(getThis());
650:     value = spl_ptr_llist_last(intern->llist);
651:
652:     if (value == NULL || Z_ISUNDEF_P(value)) {
653:         zend_throw_exception(spl_ow_RuntimeException, "Can't peek at an empty datastructure", 0);
654:         return;
655:     }
656:
657:     ZVAL_DEREF(value);
658:     ZVAL_COPY(return_value, value);
659:
660:     /* }}} */
661:
662: /* {{{ proto mixed SplDoublyLinkedList::bottom()
663: Peek at the bottom element of the SplDoublyLinkedList */
664: SPL_METHOD(SplDoublyLinkedList, bottom)
665: {
666:     zval *value;
667:     spl_dlist_object *intern;
668:
669:     if (zend_parse_parameters_none() == FAILURE) {
670:         return;
671:     }
672:
673:     intern = Z_SPDOLLIST_P(getThis());
674:     value = spl_ptr_llist_first(intern->llist);
675:
676:     if (value == NULL || Z_ISUNDEF_P(value)) {
677:         zend_throw_exception(spl_ow_RuntimeException, "Can't peek at an empty datastructure", 0);
678:         return;
679:     }
680:
681:     ZVAL_DEREF(value);
682:     ZVAL_COPY(return_value, value);
683:
684:     /* }}} */
685:
686: /* {{{ proto int SplDoublyLinkedList::count()
687: Return the number of elements in the datastructure. */
688: SPL_METHOD(SplDoublyLinkedList, count)
689: {
690:     zend_long count;
691:     spl_dlist_object *intern = Z_SPDOLLIST_P(getThis());
692:
693:     if (zend_parse_parameters_none() == FAILURE) {
694:         return;
695:     }
696:
697:     count = spl_ptr_llist_count(intern->llist);
698:     RETURN_LONG(count);
699:
700:     /* }}} */
701:
702: /* {{{ proto int SplDoublyLinkedList::isEmpty()
703: Return true if the SplDoublyLinkedList is empty. */
704: SPL_METHOD(SplDoublyLinkedList, isEmpty)
705: {
706:     zend_long count;
707:
708:     if (zend_parse_parameters_none() == FAILURE) {
709:         return;
710:     }
711:
712:     spl_dlist_object_count_elements(getThis(), &count);
713:     RETURN_BOOL(count == 0);
714:
715:     /* }}} */
716:
717: /* {{{ proto int SplDoublyLinkedList::setIteratorMode(int flags)
718: Set the mode of iteration */
719: SPL_METHOD(SplDoublyLinkedList, setIteratorMode)
720: {
721:     zend_long value;
722:     spl_dlist_object *intern;
723:
724:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &value) == FAILURE) {
725:         return;
726:     }
727:
728:     intern = Z_SPDOLLIST_P(getThis());
729:
730:     if (intern->flags & SPL_DLIST_IT_FIX) {
731:         if (intern->flags & SPL_DLIST_IT_IFPO) {
732:             zend_throw_exception(spl_ow_RuntimeException, "Iterators' LIFO/FIFO modes for SplStack/SplQueue objects are frozen", 0);
733:             return;
734:         }
735:
736:         intern->flags = (value & SPL_DLIST_IT_MASK) | (intern->flags & SPL_DLIST_IT_FIX);
737:         RETURN_LONG(intern->flags);
738:     }
739:
740:     /* }}} */
741:
742: /* {{{ proto int SplDoublyLinkedList::getIteratorMode()
743: Return the mode of iteration */
744: SPL_METHOD(SplDoublyLinkedList, getIteratorMode)
745: {
746:     spl_dlist_object *intern;
747:
748:     if (zend_parse_parameters_none() == FAILURE) {
749:         return;
750:     }
751:
752:     intern = Z_SPDOLLIST_P(getThis());

```

```
938: {
939:     SPI_LLIST_CHECK_DELREF(*traverse_pointer_ptr);
```

```
1124:     EVAL_DEREF(value);
1125:     EVAL_COPY(return_value, value);
1126: }
```

```

1127: }
1128: /* }}} */
1129:
1130: /* {{{ proto string SplDoublyLinkedList::serialize()
1131:  * Serializes storage */
1132: SPL_METHOD(SplDoublyLinkedList, serialize)
1133: {
1134:     spl_dlist_object *intern = Z_SPDLIST_P(getThis());
1135:     smart_str buf = (0);
1136:     spl_dlist_element *current = intern->llist->head, *next;
1137:     zval *zval;
1138:     php_serialize_data_t var_hash;
1139:
1140:     if (zend_parse_parameters_none() == FAILURE) {
1141:         return;
1142:     }
1143:
1144:     PHP_VAR_SERIALIZE_INIT(var_hash);
1145:
1146:     /* flags */
1147:     ZVAL_LONG(&flags, intern->flags);
1148:     php_var_serialize(&buf, &flags, &var_hash);
1149:     zval_get_ctor(&flags);
1150:
1151:     /* elements */
1152:     while (current) {
1153:         smart_str_append(&buf, ':');
1154:         next = current->next;
1155:         php_var_serialize(&buf, &current->data, &var_hash);
1156:         current = next;
1157:     }
1158:     smart_str_0(&buf);
1159:
1160:     /* done */
1161:     PHP_VAR_SERIALIZE_DESTROY(var_hash);
1162:
1163:     if (&buf.s) {
1164:         RETURN_NEW_STR(&buf.s);
1165:     } else {
1166:         RETURN_NULL();
1167:     }
1168: }
1169: /* }}} */
1170:
1171: /* {{{ proto void SplDoublyLinkedList::unserialize(string serialized)
1172:  * Unserializes storage */
1173: SPL_METHOD(SplDoublyLinkedList, unserialize)
1174: {
1175:     spl_dlist_object *intern = Z_SPDLIST_P(getThis());
1176:     zval *zval, *elem;
1177:     char *buf;
1178:     elem->buf_len;
1179:     const unsigned char *p, *s;
1180:     php_unserialize_data_t var_hash;
1181:
1182:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "s", &buf, &buf_len) == FAILURE) {
1183:         return;
1184:     }
1185:
1186:     if (&buf_len == 0) {
1187:         return;
1188:     }
1189:
1190:     a = p = (const unsigned char *)buf;
1191:     PHP_VAR_UNSERIALIZE_INIT(var_hash);
1192:
1193:     /* flags */
1194:     flags = var_tmp_var(var_hash);
1195:     if (!php_var_unserialize(&flags, &p, &buf_len, &var_hash) || Z_TYPE_P(flags) != IS_LONG) {
1196:         goto error;
1197:     }
1198:
1199:     intern->flags = (int)Z_LVAL_P(flags);
1200:
1201:     /* elements */
1202:     while (*p != '\0') {
1203:         *p;
1204:         elem = var_tmp_var(var_hash);
1205:         if (!php_var_unserialize(&elem, &p, &buf_len, &var_hash)) {
1206:             goto error;
1207:         }
1208:         var_push_ctor(&var_hash, elem);
1209:         spl_dlist_push(intern->llist, elem);
1210:     }
1211:
1212:     if (*p != '\0') {
1213:         goto error;
1214:     }
1215:
1216:     PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
1217:
1218:     return;
1219:
1220: error:
1221:     PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
1222:     zend_throw_exception(spl_ce_UnexpectedValueException, 0, "Error at offset %d of %d bytes", (&char*)p - buf, buf_len);
1223:     return;
1224: }
1225: /* }}} */
1226:
1227: /* {{{ proto void SplDoublyLinkedList::add(mixed index, mixed newval)
1228:  * Inserts a new entry before the specified index consisting of $newval. */
1229: SPL_METHOD(SplDoublyLinkedList, add)
1230: {
1231:     zval *zval, *index, *value;
1232:     spl_dlist_object *intern;
1233:     spl_dlist_element *element;
1234:     zend_long index;
1235:
1236:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "zs", &index, &value) == FAILURE) {
1237:         return;
1238:     }
1239:
1240:     intern = Z_SPDLIST_P(getThis());
1241:     index = spl_offset_convert_to_long(index);
1242:
1243:     if (index < 0 || index > intern->llist->count) {
1244:         zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid or out of range", 0);
1245:         return;
1246:     }
1247:
1248:     zval *zval;
1249:
1250:     if (index == intern->llist->count) {
1251:         spl_dlist_push(intern->llist, value);
1252:     } else {
1253:         /* Create the new element we want to insert */
1254:         spl_dlist_element *elem = emalloc(sizeof(spl_dlist_element));
1255:
1256:         /* Get the element we want to insert before */
1257:         element = spl_dlist_offset(intern->llist, index, intern->flags & SPL_DLIST_IT_LIFO);
1258:
1259:         ZVAL_COPY_VALUE(&elem->data, value);
1260:         elem->next = NULL;
1261:
1262:         /* connect to the neighbours */
1263:         if (elem->prev == NULL) {
1264:             intern->llist->head = elem;
1265:         } else {
1266:             element->prev->next = elem;
1267:         }
1268:
1269:         /* connect the neighbours to this new element */
1270:         if (elem->prev == NULL) {
1271:             intern->llist->head = elem;
1272:         } else {
1273:             element->prev->next = elem;
1274:         }
1275:
1276:         intern->llist->count++;
1277:
1278:         if (intern->llist->count == 0) {
1279:             intern->llist->ctor(elem);
1280:         }
1281:     }
1282: }
1283: /* }}} */
1284:
1285: /* {{{ iterator handler table */
1286: static const zend_object_iterator_funcs spl_dlist_it_funcs = {
1287:     spl_dlist_it_ctor,
1288:     spl_dlist_it_valid,
1289:     spl_dlist_it_get_current_data,
1290:     spl_dlist_it_get_current_key,
1291:     spl_dlist_it_move_forward,
1292:     spl_dlist_it_rewind,
1293:     NULL,
1294: };
1295:
1296: zend_object_iterator *spl_dlist_get_iterator(zend_class_entry *ce, zval *obj, int by_ref) /* {{{ */
1297: {
1298:     spl_dlist_it *iterator;
1299:     spl_dlist_object *dlist_obj = Z_SPDLIST_P(obj);
1300:
1301:     if (by_ref) {
1302:         zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1303:         return NULL;
1304:     }
1305:
1306:     iterator = emalloc(sizeof(spl_dlist_it));
1307:
1308:     zend_iterator_init(&iterator->object, iterator);
1309:
1310:     ZVAL_COPY_VALUE(&iterator->intern->it_data, obj);
1311:     iterator->intern->it_funcs = &spl_dlist_it_funcs;
1312:     iterator->intern->ce = ce;
1313:     iterator->traverse_position = dlist_obj->traverse_position;
1314:     iterator->traverse_pointer = dlist_obj->traverse_pointer;
1315:     iterator->flags = dlist_obj->flags & SPL_DLIST_IT_MASK;
1316:
1317:     ZVAL_UNDEF(iterator->intern->value);
1318:
1319:     SPL_DLIST_CHECK_ADDRESS(iterator->traverse_pointer);
1320:
1321:     return iterator->intern->it;
1322: }
1323: /* }}} */
1324:
1325: /* Functions/Class/Method definitions */
1326:
1327: ZEND_BEGIN_ARG_INFO(arginfo_spl_dlist_set_iterator_mode, 0)
1328:     ZEND_ARG_INFO(0, flags)
1329: ZEND_END_ARG_INFO()
1330:
1331: ZEND_BEGIN_ARG_INFO(arginfo_spl_dlist_push, 0)
1332:     ZEND_ARG_INFO(0, value)
1333: ZEND_END_ARG_INFO()
1334:
1335: ZEND_BEGIN_ARG_INFO(arginfo_spl_dlist_offset_get, 0, 0, 1)
1336:     ZEND_ARG_INFO(0, index)
1337:     ZEND_ARG_INFO(0, flags)
1338: ZEND_END_ARG_INFO()
1339:
1340: ZEND_BEGIN_ARG_INFO(arginfo_spl_dlist_offset_set, 0, 0, 2)
1341:     ZEND_ARG_INFO(0, index)
1342:     ZEND_ARG_INFO(0, newval)
1343: ZEND_END_ARG_INFO()
1344:
1345: ZEND_BEGIN_ARG_INFO(arginfo_spl_dlist_void, 0)
1346:     ZEND_ARG_INFO(0, serialized)
1347: ZEND_END_ARG_INFO()
1348:
1349: static const zend_function_entry spl_func_spl_dlist[] = {
1350:     SPL_FE(spl_dlist_enqueue, enqueue, SplDoublyLinkedList, push, arginfo_spl_dlist_push, ZEND_ACC_PUBLIC),
1351:     SPL_FE(spl_dlist_dequeue, dequeue, SplDoublyLinkedList, shift, arginfo_spl_dlist_void, ZEND_ACC_PUBLIC),
1352:     SPL_FE_END
1353: };
1354:
1355: static const zend_function_entry spl_func_spl_doubly_linked_list[] = {
1356:     SPL_ME(spl_doubly_linked_list_pop, arginfo_spl_dlist_void, ZEND_ACC_PUBLIC),
1357:     SPL_ME(spl_doubly_linked_list_shift, arginfo_spl_dlist_void, ZEND_ACC_PUBLIC),
1358:     SPL_ME(spl_doubly_linked_list_push, arginfo_spl_dlist_push, ZEND_ACC_PUBLIC),
1359:     SPL_ME(spl_doubly_linked_list_unshift, arginfo_spl_dlist_push, ZEND_ACC_PUBLIC),
1360:     SPL_ME(spl_doubly_linked_list_top, arginfo_spl_dlist_void, ZEND_ACC_PUBLIC),
1361:     SPL_ME(spl_doubly_linked_list_bottom, arginfo_spl_dlist_void, ZEND_ACC_PUBLIC),
1362:     SPL_ME(spl_doubly_linked_list_isempty, arginfo_spl_dlist_void, ZEND_ACC_PUBLIC),
1363:     SPL_ME(spl_doubly_linked_list_set_iterator_mode, arginfo_spl_dlist_set_iterator_mode, ZEND_ACC_PUBLIC),
1364:     SPL_ME(spl_doubly_linked_list_get_iterator_mode, arginfo_spl_dlist_void, ZEND_ACC_PUBLIC),
1365:     /* Countable */
1366:     SPL_ME(spl_doubly_linked_list_count, arginfo_spl_dlist_void, ZEND_ACC_PUBLIC),
1367:     /* ArrayAccess */
1368:     SPL_ME(spl_doubly_linked_list_offset_exists, arginfo_spl_dlist_offset_get, ZEND_ACC_PUBLIC),
1369:     SPL_ME(spl_doubly_linked_list_offset_get, arginfo_spl_dlist_offset_get, ZEND_ACC_PUBLIC),
1370:     SPL_ME(spl_doubly_linked_list_offset_set, arginfo_spl_dlist_offset_get, ZEND_ACC_PUBLIC),
1371:     SPL_ME(spl_doubly_linked_list_offset_unset, arginfo_spl_dlist_offset_get, ZEND_ACC_PUBLIC),
1372:     SPL_ME(spl_doubly_linked_list_add, arginfo_spl_dlist_offset_set, ZEND_ACC_PUBLIC),
1373:
1374:     /* Iterator */
1375:     SPL_ME(spl_doubly_linked_list_rewind, arginfo_spl_dlist_void, ZEND_ACC_PUBLIC),
1376:     SPL_ME(spl_doubly_linked_list_current, arginfo_spl_dlist_void, ZEND_ACC_PUBLIC),
1377:     SPL_ME(spl_doubly_linked_list_key, arginfo_spl_dlist_void, ZEND_ACC_PUBLIC),
1378:     SPL_ME(spl_doubly_linked_list_next, arginfo_spl_dlist_void, ZEND_ACC_PUBLIC),
1379:     SPL_ME(spl_doubly_linked_list_prev, arginfo_spl_dlist_void, ZEND_ACC_PUBLIC),
1380:     SPL_ME(spl_doubly_linked_list_valid, arginfo_spl_dlist_void, ZEND_ACC_PUBLIC),
1381:     /* Serializable */
1382:     SPL_ME(spl_doubly_linked_list_unserialize, arginfo_spl_dlist_unserialize, ZEND_ACC_PUBLIC),
1383:     SPL_ME(spl_doubly_linked_list_serialize, arginfo_spl_dlist_void, ZEND_ACC_PUBLIC),
1384:     SPL_FE_END
1385: };
1386: /* }}} */
1387:
1388: PHP_MINIT_FUNCTION(spl_dlist) /* {{{ */
1389: {
1390:     REGISTER_SPL_STO_CLASS_EX(SplDoublyLinkedList, spl_dlist_object_new, spl_func_spl_doubly_linked_list);
1391:     memory(spl_handler_SplDoublyLinkedList, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1392:
1393:     spl_handler_SplDoublyLinkedList.offset = XOffsetOf(spl_dlist_object, std);
1394:     spl_handler_SplDoublyLinkedList.clone_obj = spl_dlist_object_clone;
1395:     spl_handler_SplDoublyLinkedList.count_elements = spl_dlist_object_count_elements;
1396:     spl_handler_SplDoublyLinkedList.get_debug_info = spl_dlist_object_get_debug_info;
1397:     spl_handler_SplDoublyLinkedList.get_gc = spl_dlist_object_get_gc;
1398:     spl_handler_SplDoublyLinkedList.dtor_obj = zend_objects_destroy_object;
1399:     spl_handler_SplDoublyLinkedList.free_obj = spl_dlist_object_free_storage;
1400:
1401:     REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_LIFO", SPL_DLIST_IT_LIFO);
1402:     REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_FIFO", 0);
1403:     REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_DELETE", SPL_DLIST_IT_DELETE);
1404:     REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_KEEP", 0);
1405:
1406:     REGISTER_SPL_IMPLEMENT(SplDoublyLinkedList, Iterator);
1407:     REGISTER_SPL_IMPLEMENT(SplDoublyLinkedList, Countable);
1408:     REGISTER_SPL_IMPLEMENT(SplDoublyLinkedList, ArrayAccess);
1409:     REGISTER_SPL_IMPLEMENT(SplDoublyLinkedList, Serializable);
1410:
1411:     spl_ce_SplDoublyLinkedList->get_iterator = spl_dlist_get_iterator;
1412:
1413:     REGISTER_SPL_STO_CLASS_EX(SplQueue, SplDoublyLinkedList, spl_dlist_object_new, spl_func_spl_queue);
1414:     REGISTER_SPL_STO_CLASS_EX(SplStack, SplDoublyLinkedList, spl_dlist_object_new, NULL);
1415:
1416:     spl_ce_SplQueue->get_iterator = spl_dlist_get_iterator;
1417:     spl_ce_SplStack->get_iterator = spl_dlist_get_iterator;
1418:
1419:     return SUCCESS;
1420: }
1421: /* }}} */
1422:
1423: /*
1424:  * Local variables:
1425:  * tab-width: 4
1426:  * indent-offset: 4
1427:  * End:
1428:  * vim600: fdm=marker
1429:  * vim: noet sw=4 ts=4
1430:  */

```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: #ifdef HAVE_CONFIG_H
20: # include "config.h"
21: #endif
22:
23: #include "php.h"
24: #include "php_ini.h"
25: #include "ext/standard/info.h"
26: #include "zend_interfaces.h"
27:
28: #include "spl_spl.h"
29: #include "spl_functions.h"
30: #include "spl_engine.h"
31:
32: #include "spl_array.h"
33:
34: /* {{{ spl_initialize */
35: PHPAPI void spl_initialize(zend_class_entry *pce, zval *object)
36: {
37:     object_init_ex(object, pce);
38: }
39: /* }}} */
40:
41: PHPAPI zend_long spl_offset_convert_to_long(zval *offset) /* {{{ */
42: {
43:     zend_ulong idx;
44:
45:     try_again:
46:     switch (Z_TYPE_P(offset)) {
47:         case IS_STRING:
48:             if (ZEND_HANDLE_NUMERIC(Z_STR_P(offset), idx)) {
49:                 return idx;
50:             }
51:             break;
52:         case IS_DOUBLE:
53:             return (zend_long)Z_DVAL_P(offset);
54:         case IS_LONG:
55:             return Z_LVAL_P(offset);
56:         case IS_FALSE:
57:             return 0;
58:         case IS_TRUE:
59:             return 1;
60:         case IS_REFERENCE:
61:             offset = Z_REFVAL_P(offset);
62:             goto try_again;
63:         case IS_RESOURCE:
64:             return Z_RES_HANDLE_P(offset);
65:     }
66:     return -1;
67: }
68: /* }}} */
69:
70: /*
71:  * Local Variables:
72:  * tab-width: 4
73:  * c-basic-offset: 4
74:  * End:
75:  * vim600: fdm=marker
76:  * vim: noet sw=4 ts=4
77:  */
```

```

1: /*
2:  *
3:  * PHP Version 7
4:  *
5:  * Copyright (c) 1997-2018 The PHP Group
6:  *
7:  * This source file is subject to version 3.01 of the PHP license,
8:  * that is bundled with this package in the file LICENSE, and is
9:  * available through the world-wide-web at the following url:
10:  * http://www.php.net/license/3.01.txt
11:  * If you did not receive a copy of the PHP license and are unable to
12:  * obtain it through the world-wide-web, please send a note to
13:  * license@php.net so we can mail you a copy immediately.
14:  *
15:  * Author: Marcus Boerger <chelly@php.net>
16:  */
17:
18:
19: /* $Id$ */
20:
21: #ifndef HAVE_CONFIG_H
22: #include "config.h"
23: #endif
24:
25: #include "php.h"
26: #include "php_ini.h"
27: #include "ext/standard/info.h"
28: #include "ext/standard/file.h"
29: #include "ext/standard/php_string.h"
30: #include "zend_compile.h"
31: #include "zend_exceptions.h"
32: #include "zend_interfaces.h"
33:
34: #include "php_spl.h"
35: #include "spl_functions.h"
36: #include "spl_engine.h"
37: #include "spl_iterators.h"
38: #include "spl_directory.h"
39: #include "spl_exceptions.h"
40:
41: #include "php.h"
42: #include "fopen_wrappers.h"
43:
44: #include "ext/standard/basic_functions.h"
45: #include "ext/standard/php_filestat.h"
46:
47: #define SPL_HAS_FLAG(flags, test_flag) ((flags & test_flag) ? 1 : 0)
48:
49: /* declare the class handlers */
50: static zend_object_handlers spl_filesystem_object_handlers;
51: /* Includes handler to validate object state when retrieving methods */
52: static zend_object_handlers spl_filesystem_object_check_handlers;
53:
54: /* declare the class entry */
55: PHPAPI zend_class_entry *spl_ce_SplFileInfo;
56: PHPAPI zend_class_entry *spl_ce_DirectoryIterator;
57: PHPAPI zend_class_entry *spl_ce_FilesystemIterator;
58: PHPAPI zend_class_entry *spl_ce_CurdirIterator;
59: PHPAPI zend_class_entry *spl_ce_GlobIterator;
60: PHPAPI zend_class_entry *spl_ce_SplFileObject;
61: PHPAPI zend_class_entry *spl_ce_SplTempFileObject;
62:
63: static void spl_filesystem_file_free_line(spl_filesystem_object *intern) /* {{{ */
64: {
65:     if (intern->u.file.current_line) {
66:         efree(intern->u.file.current_line);
67:         intern->u.file.current_line = NULL;
68:     }
69:     if (IS_INDEXED(intern->u.file.current_rval)) {
70:         eval_ptr_dtor(intern->u.file.current_rval);
71:         ZVAL_UNDEF(intern->u.file.current_rval);
72:     }
73: } /* }}} */
74:
75: static void spl_filesystem_object_destroy_object(spl_filesystem_object *intern) /* {{{ */
76: {
77:     spl_filesystem_object *intern = spl_filesystem_from_obj(object);
78:
79:     zend_object_dtor(object);
80:
81:     switch (intern->type) {
82:         case SPL_FS_DIR:
83:             if (intern->u.dir.dirp) {
84:                 php_stream_close(intern->u.dir.dirp);
85:                 intern->u.dir.dirp = NULL;
86:             }
87:             break;
88:         case SPL_FS_FILE:
89:             if (intern->u.file.stream) {
90:                 /*
91:                  * If (intern->u.file.zcontext) {
92:                  *     zend_list_delref(Z_RES_VAL_P(intern->zcontext));
93:                  * }
94:                  */
95:                 if ((intern->u.file.stream->is_persistent) &
96:                     !php_stream_close(intern->u.file.stream)) {
97:                     /*
98:                      * else {
99:                      *     php_stream_close(intern->u.file.stream);
100:                      * }
101:                     */
102:                     break;
103:                 default:
104:                     break;
105:             } /* }}} */
106:
107: static void spl_filesystem_object_free_storage(zend_object *object) /* {{{ */
108: {
109:     spl_filesystem_object *intern = spl_filesystem_from_obj(object);
110:
111:     if (intern->both_handler && intern->both_handler->dtor) {
112:         intern->both_handler->dtor(intern);
113:     }
114:
115:     zend_object_std_dtor(intern->std);
116:
117:     if (intern->u.path) {
118:         efree(intern->u.path);
119:     }
120:     if (intern->u.file_name) {
121:         efree(intern->u.file_name);
122:     }
123:     switch (intern->type) {
124:         case SPL_FS_DIR:
125:             break;
126:         case SPL_FS_FILE:
127:             if (intern->u.dir.sub_path) {
128:                 efree(intern->u.dir.sub_path);
129:             }
130:             break;
131:         case SPL_FS_FILE:
132:             if (intern->u.file.stream) {
133:                 if (intern->u.file.open_mode) {
134:                     efree(intern->u.file.open_mode);
135:                 }
136:                 if (intern->u.orig_path) {
137:                     efree(intern->u.orig_path);
138:                 }
139:             }
140:             break;
141:         case SPL_FS_FILE:
142:             break;
143:     } /* }}} */
144:
145: /* {{{ spl_ce_dir_object_new */
146: /* creates the object by
147:  * - allocating memory
148:  * - initializing the object members
149:  * - storing the object
150:  * - setting its handlers
151:  */
152: /* called from
153:  * - clone
154:  * - new
155:  */
156: static zend_object *spl_filesystem_object_new_ex(zend_class_entry *class_type)
157: {
158:     spl_filesystem_object *intern;
159:
160:     intern = zend_object_alloc(sizeof(spl_filesystem_object), class_type);
161:     /* intern->type = SPL_FS_DIR; done by set 0 */
162:     intern->file_class = spl_ce_SplFileObject;
163:     intern->info_class = spl_ce_SplFileInfo;
164:
165:     zend_object_std_init(intern->std, class_type);
166:     object_properties_init(intern->std, class_type);
167:     intern->std.handlers = spl_filesystem_object_handlers;
168:
169:     return intern->std;
170: } /* }}} */
171:
172:
173: /* {{{ spl_filesystem_object_new */
174: /* See spl_filesystem_object_new_ex */
175: static zend_object *spl_filesystem_object_new(zend_class_entry *class_type)
176: {
177:     return spl_filesystem_object_new_ex(class_type);
178: }
179:
180:
181: /* {{{ spl_filesystem_object_new_check */
182: static zend_object *spl_filesystem_object_new_check(zend_class_entry *class_type)
183: {
184:     spl_filesystem_object *ret = spl_filesystem_from_obj(spl_filesystem_object_new_ex(class_type));
185:     ret->std.handlers = spl_filesystem_object_check_handlers;
186:     return ret->std;
187: }
188:
189:
190:
191: PHPAPI char *spl_filesystem_object_get_path(spl_filesystem_object *intern, size_t *len) /* {{{ */
192: {
193:     if (intern->type == SPL_FS_DIR) {
194:         if (php_stream_is(intern->u.dir.dirp, &php_glob_stream_ops)) {
195:             return php_glob_stream_get_path(intern->u.dir.dirp, 0, len);
196:         }
197:     }
198:     if (len) {
199:         *len = intern->u.path_len;
200:     }
201:     return intern->u.path;
202: } /* }}} */
203:
204:
205: static inline void spl_filesystem_object_get_file_name(spl_filesystem_object *intern) /* {{{ */
206: {
207:     char slash = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_UNIPATHS) ? '/' : DEFAULT_SLASH;
208:
209:     switch (intern->type) {
210:         case SPL_FS_DIR:
211:             if (intern->u.file_name) {
212:                 php_error_docref(NULL, E_ERROR, "Object not initialized");
213:             }
214:             break;
215:         case SPL_FS_FILE:
216:             if (intern->u.file_name) {
217:                 efree(intern->u.file_name);
218:             }
219:             intern->u.file_name_len = spprintf(&intern->u.file_name, 0, "%s%s",
220:                 spl_filesystem_object_get_path(intern, NULL),
221:                 slash, intern->u.dir.entry_d_name);
222:             break;
223:     }
224: } /* }}} */
225:
226:
227: static int spl_filesystem_dir_read(spl_filesystem_object *intern) /* {{{ */
228: {
229:     if ((intern->u.dir.dirp || !php_stream_readdir(intern->u.dir.dirp, &intern->u.dir.entry)) &
230:         !intern->u.dir.entry_d_name[0] == '\0') {
231:         return 0;
232:     } else {
233:         return 1;
234:     }
235: } /* }}} */
236:
237:
238: #define IS_SLASH_AT(pos, pos) (IS_SLASH(pos[pos]))
239:
240: static inline int spl_filesystem_is_dot(const char *d_name) /* {{{ */
241: {
242:     return !strcmp(d_name, ".") || !strcmp(d_name, "..");
243: } /* }}} */
244:
245:
246: /* {{{ spl_filesystem_dir_open */
247: /* open a directory resource */
248: static void spl_filesystem_dir_open(spl_filesystem_object *intern, char *path)
249: {
250:     int skip_dots = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_SKIPDOTS);
251:
252:     intern->type = SPL_FS_DIR;
253:     intern->u.path_len = strlen(path);
254:     intern->u.dir.dirp = php_stream_opendir(path, REPORT_ERRORS, PG(default_context));
255:     if (intern->u.path_len > 1 && IS_SLASH_AT(path, intern->u.path_len-1)) {
256:         intern->u.path = estrndup(path, intern->u.path_len);
257:     } else {
258:         intern->u.path = estrndup(path, intern->u.path_len);
259:     }
260:     intern->u.dir.index = 0;
261:
262:     if (EG(exception)) || intern->u.dir.dirp == NULL ||
263:         !intern->u.dir.entry_d_name[0] == '\0' ||
264:         !EG(exception)) {
265:             /* open failed w/out notice (turned to exception due to E_THROW) */
266:             zend_throw_exception(spl_ce_RuntimeException, 0, "Failed to open directory \"%s\".", path);
267:         } else {
268:             /*
269:              * do {
270:              *     spl_filesystem_dir_read(intern);
271:              * } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
272:              */
273:         } /* }}} */
274:
275:     static int spl_filesystem_file_open(spl_filesystem_object *intern, int use_include_path, int silent) /* {{{ */
276:     {
277:         zend_tmp;
278:
279:         intern->type = SPL_FS_FILE;
280:
281:         php_stat(intern->u.file_name, intern->u.file_name_len, FS_IS_DIR, &tmp);
282:         if (IS_TYPE(tmp) == IS_THROW) {
283:             intern->u.file.open_mode = NULL;
284:             intern->u.file_name = NULL;
285:             zend_throw_exception(spl_ce_LogicException, 0, "Cannot use SplFileObject with directories");
286:             return FAILURE;
287:         }
288:
289:         intern->u.file.zcontext = php_stream_context_from_rval(intern->u.file.zcontext, 0);
290:         intern->u.file.stream = php_stream_open_wrapper_ex(intern->u.file_name, intern->u.file.open_mode, (use_include_path ? USE_PATH : 0) | REPORT_ERRORS, MU
291:             L, intern->u.file.context);
292:
293:         if (intern->u.file_name_len != !intern->u.file.stream) {
294:             if (EG(exception)) {
295:                 zend_throw_exception(spl_ce_RuntimeException, 0, "Cannot open file \"%s\", intern->u.file_name_len ? intern->u.file_name : "");
296:             }
297:             intern->u.file_name = NULL; /* until here it is not a copy */
298:             intern->u.file.open_mode = NULL;
299:             return FAILURE;
300:         }
301:
302:         /*
303:          * If (intern->u.file.zcontext) {
304:          *     //zend_list_addref(Z_RES_VAL(intern->u.file.zcontext));
305:          *     Z_ADDREF_P(intern->u.file.zcontext);
306:          * }
307:          */
308:
309:         if (intern->u.file_name_len > 1 && IS_SLASH_AT(intern->u.file_name, intern->u.file_name_len-1)) {
310:             intern->u.file_name_len--;
311:         }
312:
313:         intern->u.orig_path = estrndup(intern->u.file.stream->orig_path, strlen(intern->u.file.stream->orig_path));
314:
315:         intern->u.file_name = estrndup(intern->u.file_name, intern->u.file_name_len);
316:         intern->u.file.open_mode = estrndup(intern->u.file.open_mode, intern->u.file.open_mode_len);
317:
318:         /* avoid reference counting in debug mode, thus do it manually */
319:         ZVAL_RES(intern->u.file.resource, intern->u.file.stream->res);
320:
321:         /*!!! TODO: maybe bug?
322:          * Z_SET_REFCOUNT(intern->u.file.resource, 1);
323:          */
324:
325:         intern->u.file.delimiter = '/';
326:         intern->u.file.enclosure = '';
327:         intern->u.file.escape = '\\';
328:
329:         intern->u.file.func_getCurr = zend_hash_str_find_ptr(intern->std.ce->function_table, "getcurrenttime", sizeof("getcurrenttime") - 1);
330:
331:         return SUCCESS;
332:     } /* }}} */
333:
334:
335: /* {{{ spl_filesystem_object_clone */
336: /* Local zend_object creation (on stack)
337:  * Load the 'other' object
338:  * Create a new empty object (See spl_filesystem_object_new_ex)
339:  * Open the directory
340:  * Clone other members (properties)
341:  */
342: static zend_object *spl_filesystem_object_clone(zval *obj)
343: {
344:     zend_object *old_obj;
345:     zend_object *new_obj;
346:     spl_filesystem_object *intern;
347:     spl_filesystem_object *source;
348:     int index, skip_dots;
349:
350:     old_obj = Z_OBJ_P(obj);
351:     source = spl_filesystem_from_obj(old_obj);
352:     new_obj = spl_filesystem_object_new_ex(old_obj->ce);
353:     intern = spl_filesystem_from_obj(new_obj);
354:
355:     intern->flags = source->flags;
356:
357:     switch (source->type) {
358:         case SPL_FS_DIR:
359:             intern->u.path_len = source->u.path_len;
360:             intern->u.path = estrndup(source->u.path, source->u.path_len);
361:             intern->u.file_name_len = source->u.file_name_len;
362:             intern->u.file_name = estrndup(source->u.file_name, intern->u.file_name_len);
363:             break;
364:         case SPL_FS_FILE:
365:             spl_filesystem_dir_open(intern, source->u.path);
366:             /* read until we hit the position in which we were before */
367:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
368:             for (index = 0; index < source->u.dir.index; ++index) {
369:                 do {
370:                     spl_filesystem_dir_read(intern);
371:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
372:             }
373:             intern->u.dir.index = index;
374:             break;
375:         case SPL_FS_FILE:
376:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
377:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
378:             for (index = 0; index < source->u.dir.index; ++index) {
379:                 do {
380:                     spl_filesystem_dir_read(intern);
381:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
382:             }
383:             intern->u.dir.index = index;
384:             break;
385:         case SPL_FS_FILE:
386:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
387:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
388:             for (index = 0; index < source->u.dir.index; ++index) {
389:                 do {
390:                     spl_filesystem_dir_read(intern);
391:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
392:             }
393:             intern->u.dir.index = index;
394:             break;
395:         case SPL_FS_FILE:
396:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
397:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
398:             for (index = 0; index < source->u.dir.index; ++index) {
399:                 do {
400:                     spl_filesystem_dir_read(intern);
401:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
402:             }
403:             intern->u.dir.index = index;
404:             break;
405:         case SPL_FS_FILE:
406:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
407:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
408:             for (index = 0; index < source->u.dir.index; ++index) {
409:                 do {
410:                     spl_filesystem_dir_read(intern);
411:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
412:             }
413:             intern->u.dir.index = index;
414:             break;
415:         case SPL_FS_FILE:
416:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
417:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
418:             for (index = 0; index < source->u.dir.index; ++index) {
419:                 do {
420:                     spl_filesystem_dir_read(intern);
421:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
422:             }
423:             intern->u.dir.index = index;
424:             break;
425:         case SPL_FS_FILE:
426:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
427:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
428:             for (index = 0; index < source->u.dir.index; ++index) {
429:                 do {
430:                     spl_filesystem_dir_read(intern);
431:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
432:             }
433:             intern->u.dir.index = index;
434:             break;
435:         case SPL_FS_FILE:
436:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
437:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
438:             for (index = 0; index < source->u.dir.index; ++index) {
439:                 do {
440:                     spl_filesystem_dir_read(intern);
441:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
442:             }
443:             intern->u.dir.index = index;
444:             break;
445:         case SPL_FS_FILE:
446:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
447:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
448:             for (index = 0; index < source->u.dir.index; ++index) {
449:                 do {
450:                     spl_filesystem_dir_read(intern);
451:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
452:             }
453:             intern->u.dir.index = index;
454:             break;
455:         case SPL_FS_FILE:
456:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
457:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
458:             for (index = 0; index < source->u.dir.index; ++index) {
459:                 do {
460:                     spl_filesystem_dir_read(intern);
461:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
462:             }
463:             intern->u.dir.index = index;
464:             break;
465:         case SPL_FS_FILE:
466:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
467:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
468:             for (index = 0; index < source->u.dir.index; ++index) {
469:                 do {
470:                     spl_filesystem_dir_read(intern);
471:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
472:             }
473:             intern->u.dir.index = index;
474:             break;
475:         case SPL_FS_FILE:
476:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
477:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
478:             for (index = 0; index < source->u.dir.index; ++index) {
479:                 do {
480:                     spl_filesystem_dir_read(intern);
481:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
482:             }
483:             intern->u.dir.index = index;
484:             break;
485:         case SPL_FS_FILE:
486:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
487:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
488:             for (index = 0; index < source->u.dir.index; ++index) {
489:                 do {
490:                     spl_filesystem_dir_read(intern);
491:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
492:             }
493:             intern->u.dir.index = index;
494:             break;
495:         case SPL_FS_FILE:
496:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
497:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
498:             for (index = 0; index < source->u.dir.index; ++index) {
499:                 do {
500:                     spl_filesystem_dir_read(intern);
501:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
502:             }
503:             intern->u.dir.index = index;
504:             break;
505:         case SPL_FS_FILE:
506:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
507:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
508:             for (index = 0; index < source->u.dir.index; ++index) {
509:                 do {
510:                     spl_filesystem_dir_read(intern);
511:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
512:             }
513:             intern->u.dir.index = index;
514:             break;
515:         case SPL_FS_FILE:
516:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
517:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
518:             for (index = 0; index < source->u.dir.index; ++index) {
519:                 do {
520:                     spl_filesystem_dir_read(intern);
521:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
522:             }
523:             intern->u.dir.index = index;
524:             break;
525:         case SPL_FS_FILE:
526:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
527:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
528:             for (index = 0; index < source->u.dir.index; ++index) {
529:                 do {
530:                     spl_filesystem_dir_read(intern);
531:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
532:             }
533:             intern->u.dir.index = index;
534:             break;
535:         case SPL_FS_FILE:
536:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
537:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
538:             for (index = 0; index < source->u.dir.index; ++index) {
539:                 do {
540:                     spl_filesystem_dir_read(intern);
541:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
542:             }
543:             intern->u.dir.index = index;
544:             break;
545:         case SPL_FS_FILE:
546:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
547:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
548:             for (index = 0; index < source->u.dir.index; ++index) {
549:                 do {
550:                     spl_filesystem_dir_read(intern);
551:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
552:             }
553:             intern->u.dir.index = index;
554:             break;
555:         case SPL_FS_FILE:
556:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
557:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
558:             for (index = 0; index < source->u.dir.index; ++index) {
559:                 do {
560:                     spl_filesystem_dir_read(intern);
561:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
562:             }
563:             intern->u.dir.index = index;
564:             break;
565:         case SPL_FS_FILE:
566:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
567:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
568:             for (index = 0; index < source->u.dir.index; ++index) {
569:                 do {
570:                     spl_filesystem_dir_read(intern);
571:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
572:             }
573:             intern->u.dir.index = index;
574:             break;
575:         case SPL_FS_FILE:
576:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
577:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
578:             for (index = 0; index < source->u.dir.index; ++index) {
579:                 do {
580:                     spl_filesystem_dir_read(intern);
581:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
582:             }
583:             intern->u.dir.index = index;
584:             break;
585:         case SPL_FS_FILE:
586:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
587:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
588:             for (index = 0; index < source->u.dir.index; ++index) {
589:                 do {
590:                     spl_filesystem_dir_read(intern);
591:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
592:             }
593:             intern->u.dir.index = index;
594:             break;
595:         case SPL_FS_FILE:
596:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
597:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
598:             for (index = 0; index < source->u.dir.index; ++index) {
599:                 do {
600:                     spl_filesystem_dir_read(intern);
601:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
602:             }
603:             intern->u.dir.index = index;
604:             break;
605:         case SPL_FS_FILE:
606:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
607:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
608:             for (index = 0; index < source->u.dir.index; ++index) {
609:                 do {
610:                     spl_filesystem_dir_read(intern);
611:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
612:             }
613:             intern->u.dir.index = index;
614:             break;
615:         case SPL_FS_FILE:
616:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
617:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
618:             for (index = 0; index < source->u.dir.index; ++index) {
619:                 do {
620:                     spl_filesystem_dir_read(intern);
621:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
622:             }
623:             intern->u.dir.index = index;
624:             break;
625:         case SPL_FS_FILE:
626:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
627:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
628:             for (index = 0; index < source->u.dir.index; ++index) {
629:                 do {
630:                     spl_filesystem_dir_read(intern);
631:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
632:             }
633:             intern->u.dir.index = index;
634:             break;
635:         case SPL_FS_FILE:
636:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
637:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
638:             for (index = 0; index < source->u.dir.index; ++index) {
639:                 do {
640:                     spl_filesystem_dir_read(intern);
641:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
642:             }
643:             intern->u.dir.index = index;
644:             break;
645:         case SPL_FS_FILE:
646:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
647:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
648:             for (index = 0; index < source->u.dir.index; ++index) {
649:                 do {
650:                     spl_filesystem_dir_read(intern);
651:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
652:             }
653:             intern->u.dir.index = index;
654:             break;
655:         case SPL_FS_FILE:
656:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
657:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
658:             for (index = 0; index < source->u.dir.index; ++index) {
659:                 do {
660:                     spl_filesystem_dir_read(intern);
661:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
662:             }
663:             intern->u.dir.index = index;
664:             break;
665:         case SPL_FS_FILE:
666:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
667:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
668:             for (index = 0; index < source->u.dir.index; ++index) {
669:                 do {
670:                     spl_filesystem_dir_read(intern);
671:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
672:             }
673:             intern->u.dir.index = index;
674:             break;
675:         case SPL_FS_FILE:
676:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
677:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
678:             for (index = 0; index < source->u.dir.index; ++index) {
679:                 do {
680:                     spl_filesystem_dir_read(intern);
681:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
682:             }
683:             intern->u.dir.index = index;
684:             break;
685:         case SPL_FS_FILE:
686:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
687:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
688:             for (index = 0; index < source->u.dir.index; ++index) {
689:                 do {
690:                     spl_filesystem_dir_read(intern);
691:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
692:             }
693:             intern->u.dir.index = index;
694:             break;
695:         case SPL_FS_FILE:
696:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
697:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
698:             for (index = 0; index < source->u.dir.index; ++index) {
699:                 do {
700:                     spl_filesystem_dir_read(intern);
701:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
702:             }
703:             intern->u.dir.index = index;
704:             break;
705:         case SPL_FS_FILE:
706:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
707:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
708:             for (index = 0; index < source->u.dir.index; ++index) {
709:                 do {
710:                     spl_filesystem_dir_read(intern);
711:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
712:             }
713:             intern->u.dir.index = index;
714:             break;
715:         case SPL_FS_FILE:
716:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
717:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
718:             for (index = 0; index < source->u.dir.index; ++index) {
719:                 do {
720:                     spl_filesystem_dir_read(intern);
721:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
722:             }
723:             intern->u.dir.index = index;
724:             break;
725:         case SPL_FS_FILE:
726:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
727:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
728:             for (index = 0; index < source->u.dir.index; ++index) {
729:                 do {
730:                     spl_filesystem_dir_read(intern);
731:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
732:             }
733:             intern->u.dir.index = index;
734:             break;
735:         case SPL_FS_FILE:
736:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
737:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
738:             for (index = 0; index < source->u.dir.index; ++index) {
739:                 do {
740:                     spl_filesystem_dir_read(intern);
741:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
742:             }
743:             intern->u.dir.index = index;
744:             break;
745:         case SPL_FS_FILE:
746:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
747:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
748:             for (index = 0; index < source->u.dir.index; ++index) {
749:                 do {
750:                     spl_filesystem_dir_read(intern);
751:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
752:             }
753:             intern->u.dir.index = index;
754:             break;
755:         case SPL_FS_FILE:
756:             spl_filesystem_file_open(intern, source->u.use_include_path, source->u.silent);
757:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
758:             for (index = 0; index < source->u.dir.index; ++index) {
759:                 do {
760:                     spl_filesystem_dir_read(intern);
761:                 } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name));
762:             }
763:             intern->u.dir.index = index;
764:            
```

```

376:     zend_throw_error(NULL, "An object of class is cannot be cloned", ZSTR_VAL(oid_object->ce->name));
377:     return new_object;
378: }
379:
380: intern->file_class = source->file_class;
381: intern->info_class = source->info_class;
382: intern->path = source->path;
383: intern->path_handler = source->path_handler;
384:
385: zend_object_clone_members(new_object, oid_object);
386:
387: IF (intern->path_handler && intern->path_handler->clone) {
388:     intern->path_handler->clone(source, intern);
389: }
390:
391: return new_object;
392: }
393: /* }}} */
394:
395: void spl_filesystem_info_set_filename(spl_filesystem_object *intern, char *path, size_t len, size_t use_copy) /* {{{ */
396: {
397:     char *p1, *p2;
398:
399:     IF (intern->file_name) {
400:         efree(intern->file_name);
401:     }
402:
403:     intern->file_name = use_copy ? estrndup(path, len) : path;
404:     intern->file_name_len = len;
405:
406:     while (intern->file_name_len > 1 && IS_SLASH_AT(intern->file_name, intern->file_name_len-1)) {
407:         intern->file_name[intern->file_name_len-1] = 0;
408:         intern->file_name_len--;
409:     }
410:
411:     p1 = strchr(intern->file_name, '/');
412:     IF (defined(PHP_WIN32))
413:         p2 = strchr(intern->file_name, '\\');
414:     else
415:         p2 = 0;
416:     IF (p1 || p2) {
417:         intern->path_len = ((p1 > p2 ? p1 : p2) - intern->file_name);
418:     } else {
419:         intern->path_len = 0;
420:     }
421:
422:     IF (intern->path) {
423:         efree(intern->path);
424:     }
425:
426:     intern->path = estrndup(path, intern->path_len);
427: } /* }}} */
428:
429: static spl_filesystem_object *spl_filesystem_object_create_info(spl_filesystem_object *source, char *file_name, size_t file_path_len, int use_copy, zend_class_entry *ce, zval *return_value) /* {{{ */
430: {
431:     spl_filesystem_object *intern;
432:     zval arg1;
433:     zend_error_handling error_handling;
434:
435:     IF (file_path || file_path_len) {
436:         IF (defined(PHP_WIN32))
437:             zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot create SplFileInfo for empty path");
438:         IF (file_path && !use_copy) {
439:             efree(file_path);
440:         }
441:         else
442:             IF (file_path && !use_copy) {
443:                 efree(file_path);
444:             }
445:             file_path_len = 1;
446:             file_path = "/";
447:         IF (defined(PHP_WIN32))
448:             return NULL;
449:     }
450:
451:     zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, error_handling);
452:
453:     ce = ce ? ce : source->info_class;
454:
455:     zend_update_class_constants(ce);
456:
457:     intern = spl_filesystem_from_obj(spl_filesystem_object_new_ex(ce));
458:     ZVAL_OBJ(return_value, &intern->std);
459:
460:     IF (ce->constructor->common.scope != spl_ce_SplFileInfo) {
461:         ZVAL_STRING(&arg1, file_path, file_path_len);
462:         zend_call_method_with_1_param(return_value, ce, ce->constructor, "_construct", NULL, &arg1);
463:         zval_ptr_dtor(&arg1);
464:     } else {
465:         spl_filesystem_info_set_filename(intern, file_path, file_path_len, use_copy);
466:     }
467:
468:     zend_restore_error_handling(error_handling);
469:     return intern;
470: } /* }}} */
471:
472: static spl_filesystem_object *spl_filesystem_object_create_type(int ht, spl_filesystem_object *source, int type, zend_class_entry *ce, zval *return_value) /* {{{ */
473: {
474:     spl_filesystem_object *intern;
475:     zend_bool use_include_path = 0;
476:     zval arg1, arg2;
477:     zend_error_handling error_handling;
478:
479:     zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, error_handling);
480:
481:     switch (source->type) {
482:         case SPL_FS_THROW:
483:             case SPL_FS_FILE:
484:                 break;
485:             case SPL_FS_DIR:
486:                 IF (!source->u.dir.entry.d_name[0]) {
487:                     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Could not open file");
488:                     zend_restore_error_handling(error_handling);
489:                     return NULL;
490:                 }
491:             }
492:
493:     switch (type) {
494:         case SPL_FS_THROW:
495:             ce = ce ? ce : source->info_class;
496:
497:             IF (UNEXPECTED(zend_update_class_constants(ce) != SUCCESS)) {
498:                 break;
499:             }
500:
501:             intern = spl_filesystem_from_obj(spl_filesystem_object_new_ex(ce));
502:             ZVAL_OBJ(return_value, &intern->std);
503:
504:             spl_filesystem_object_get_file_name(source);
505:             IF (ce->constructor->common.scope != spl_ce_SplFileInfo) {
506:                 ZVAL_STRING(&arg1, source->file_name, source->file_name_len);
507:                 zend_call_method_with_1_param(return_value, ce, ce->constructor, "_construct", NULL, &arg1);
508:                 zval_ptr_dtor(&arg1);
509:             } else {
510:                 intern->file_name = estrndup(source->file_name, source->file_name_len);
511:                 intern->file_name_len = source->file_name_len;
512:                 intern->path = spl_filesystem_object_get_path(source, &intern->path_len);
513:                 intern->path = estrndup(intern->path, intern->path_len);
514:             }
515:             break;
516:             case SPL_FS_FILE:
517:                 ce = ce ? ce : source->file_class;
518:
519:                 IF (UNEXPECTED(zend_update_class_constants(ce) != SUCCESS)) {
520:                     break;
521:                 }
522:
523:                 intern = spl_filesystem_from_obj(spl_filesystem_object_new_ex(ce));
524:                 ZVAL_OBJ(return_value, &intern->std);
525:
526:                 spl_filesystem_object_get_file_name(source);
527:
528:                 IF (ce->constructor->common.scope != spl_ce_SplFileInfo) {
529:                     ZVAL_STRING(&arg1, source->file_name, source->file_name_len);
530:                     ZVAL_STRING(&arg2, "r", 1);
531:                     zend_call_method_with_2_param(return_value, ce, ce->constructor, "_construct", NULL, &arg1, &arg2);
532:                     zval_ptr_dtor(&arg1);
533:                     zval_ptr_dtor(&arg2);
534:                 } else {
535:                     intern->file_name = source->file_name;
536:                     intern->file_name_len = source->file_name_len;
537:                     intern->path = spl_filesystem_object_get_path(source, &intern->path_len);
538:                     intern->path = estrndup(intern->path, intern->path_len);
539:
540:                     intern->u.file.open_mode = "r";
541:                     intern->u.file.open_mode_len = 1;
542:
543:                     IF (ht && zend_parse_parameters(ht, "dbr",
544:                         &intern->u.file.open_mode,
545:                         &use_include_path,
546:                         &intern->u.file.constructor == FAILURE)) {
547:                         zend_restore_error_handling(error_handling);
548:                         intern->u.file.open_mode = NULL;
549:                         intern->file_name = NULL;
550:                         zval_ptr_dtor(return_value);
551:                         ZVAL_NULL(return_value);
552:                         return NULL;
553:                     }
554:
555:                     IF (spl_filesystem_file_open(intern, use_include_path, 0) == FAILURE) {
556:                         zend_restore_error_handling(error_handling);
557:                         zval_ptr_dtor(return_value);
558:                         ZVAL_NULL(return_value);
559:                         return NULL;
560:                     }
561: }

```

```

562:     break;
563:     case SPL_FS_DIR:
564:         zend_restore_error_handling(error_handling);
565:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Operation not supported");
566:         return NULL;
567:     }
568:     zend_restore_error_handling(error_handling);
569:     return NULL;
570: } /* }}} */
571:
572: static int spl_filesystem_is_invalid_or_dot(const char *d_name) /* {{{ */
573: {
574:     return d_name[0] == '\0' || spl_filesystem_is_dot(d_name);
575: }
576: /* }}} */
577:
578: static char *spl_filesystem_object_get_pathname(spl_filesystem_object *intern, size_t *len) /* {{{ */
579: {
580:     switch (intern->type) {
581:         case SPL_FS_THROW:
582:             case SPL_FS_FILE:
583:                 *len = intern->file_name_len;
584:                 return intern->file_name;
585:             case SPL_FS_DIR:
586:                 IF (intern->u.dir.entry.d_name[0]) {
587:                     spl_filesystem_object_get_file_name(intern);
588:                     *len = intern->file_name_len;
589:                     return intern->file_name;
590:                 }
591:                 *len = 0;
592:                 return NULL;
593:             }
594:     /* }}} */
595:
596:     static HashTable *spl_filesystem_object_get_debug_info(zval *object, int *is_temp) /* {{{ */
597:     {
598:         spl_filesystem_object *intern = Z_SPF_FILESYSTEM_P(object);
599:         zval tmp;
600:         HashTable *rv;
601:         zend_string *pstr;
602:         char *path;
603:         size_t path_len;
604:         char tmp[2];
605:
606:         *is_temp = 1;
607:
608:         IF (!intern->std.properties) {
609:             rebuild_object_properties(&intern->std);
610:         }
611:         rv = zend_array_dup(intern->std.properties);
612:
613:         pstr = spl_gen_private_prop_name(spl_ce_SplFileInfo, "pathName", sizeof("pathName")-1);
614:         path = spl_filesystem_object_get_pathname(intern, &path_len);
615:         ZVAL_STRING(&tmp, path, path_len);
616:         zend_symtable_update(rv, pstr, &tmp);
617:         zend_string_release(pstr);
618:
619:         IF (intern->file_name) {
620:             pstr = spl_gen_private_prop_name(spl_ce_SplFileInfo, "fileName", sizeof("fileName")-1);
621:             spl_filesystem_object_get_path(intern, &path_len);
622:
623:             IF (path_len && path_len < intern->file_name_len) {
624:                 ZVAL_STRING(&tmp, intern->file_name + path_len + 1, intern->file_name_len - (path_len + 1));
625:             } else {
626:                 ZVAL_STRING(&tmp, intern->file_name, intern->file_name_len);
627:             }
628:             zend_symtable_update(rv, pstr, &tmp);
629:             zend_string_release(pstr);
630:         }
631:
632:         IF (intern->type == SPL_FS_DIR) {
633:             #ifdef HAVE_GLOB
634:             pstr = spl_gen_private_prop_name(spl_ce_DirectoryIterator, "glob", sizeof("glob")-1);
635:             IF (spl_gen_stream_is(intern->u.dir.dir, &spl_glob_stream_ops)) {
636:                 ZVAL_STRING(&tmp, intern->path, intern->path_len);
637:             } else {
638:                 ZVAL_FALSE(&tmp);
639:             }
640:             zend_symtable_update(rv, pstr, &tmp);
641:             zend_string_release(pstr);
642:         }
643:         IF (pstr = spl_gen_private_prop_name(spl_ce_RecursiveDirectoryIterator, "subPathName", sizeof("subPathName")-1);
644:             IF (intern->u.dir.sub_path) {
645:                 ZVAL_STRING(&tmp, intern->u.dir.sub_path, intern->u.dir.sub_path_len);
646:             } else {
647:                 ZVAL_EMPTY_STRING(&tmp);
648:             }
649:             zend_symtable_update(rv, pstr, &tmp);
650:             zend_string_release(pstr);
651:         }
652:
653:         IF (intern->type == SPL_FS_FILE) {
654:             pstr = spl_gen_private_prop_name(spl_ce_SplFileInfo, "openMode", sizeof("openMode")-1);
655:             ZVAL_STRING(&tmp, intern->u.file.open_mode, intern->u.file.open_mode_len);
656:             zend_symtable_update(rv, pstr, &tmp);
657:             zend_string_release(pstr);
658:             tmp[0] = '\0';
659:             pstr = spl_gen_private_prop_name(spl_ce_SplFileInfo, "delimiter", sizeof("delimiter")-1);
660:             ZVAL_STRING(&tmp, &tmp, 1);
661:             zend_symtable_update(rv, pstr, &tmp);
662:             zend_string_release(pstr);
663:             tmp[0] = intern->u.file.enclosure;
664:             pstr = spl_gen_private_prop_name(spl_ce_SplFileInfo, "enclosure", sizeof("enclosure")-1);
665:             ZVAL_STRING(&tmp, &tmp, 1);
666:             zend_symtable_update(rv, pstr, &tmp);
667:             zend_string_release(pstr);
668:         }
669:
670:         return rv;
671:     } /* }}} */
672:
673:     static zend_function *spl_filesystem_object_get_method_check(zend_object **object, zend_string *method, const zval *key) /* {{{ */
674:     {
675:         spl_filesystem_object *fsobj = spl_filesystem_from_obj(*object);
676:
677:         IF (fsobj->u.dir.dir == NULL && fsobj->u.dir.path == NULL) {
678:             zend_function *func;
679:             zend_string *tmp = zend_string_init("bad_state_ex", sizeof("bad_state_ex") - 1, 0);
680:             func = zend_get_std_object_handlers()->get_method(object, tmp, NULL);
681:             zend_string_release(tmp);
682:             return func;
683:         }
684:
685:         return zend_get_std_object_handlers()->get_method(object, method, key);
686:     } /* }}} */
687:
688:     #define DT_CTOR_FLAGS 0x00000001
689:     #define DT_CTOR_GLOB 0x00000002
690:
691:     void spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAMETERS, zend_long ctor_flags) /* {{{ */
692:     {
693:         spl_filesystem_object *intern;
694:         char *path;
695:         int param;
696:         size_t len;
697:         zend_long flags;
698:         zend_error_handling error_handling;
699:
700:         zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, error_handling);
701:
702:         IF (SPL_BAS_FLAG(ctor_flags, DT_CTOR_FLAGS)) {
703:             flags = SPL_FILE_DIR_KEY_AS_PATHNAME|SPL_FILE_DIR_CURRENT_AS_FILEINFO;
704:             parsed = zend_parse_parameters(ZEND_NUM_ARGS(), "s|l", &path, &len, &flags);
705:         } else {
706:             flags = SPL_FILE_DIR_KEY_AS_PATHNAME|SPL_FILE_DIR_CURRENT_AS_SELF;
707:             parsed = zend_parse_parameters(ZEND_NUM_ARGS(), "s", &path, &len);
708:         }
709:
710:         IF (SPL_BAS_FLAG(ctor_flags, SPL_FILE_DIR_SKIPROOTS)) {
711:             flags |= SPL_FILE_DIR_SKIPROOTS;
712:         }
713:
714:         IF (SPL_BAS_FLAG(ctor_flags, SPL_FILE_DIR_UNIXPATHS)) {
715:             flags |= SPL_FILE_DIR_UNIXPATHS;
716:         }
717:
718:         IF (parsed == FAILURE) {
719:             zend_restore_error_handling(error_handling);
720:             return;
721:         }
722:
723:         IF (len) {
724:             zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Directory name must not be empty.");
725:             zend_restore_error_handling(error_handling);
726:             return;
727:         }
728:
729:         intern = Z_SPF_FILESYSTEM_P(getThis());
730:         IF (intern->path) {
731:             /* object is already initialized */
732:             zend_restore_error_handling(error_handling);
733:             php_error_docref(NULL, E_WARNING, "Directory object is already initialized");
734:             return;
735:         }
736:
737:         intern->flags = flags;
738:         #ifdef HAVE_GLOB
739:         IF (SPL_BAS_FLAG(ctor_flags, DT_CTOR_GLOB) && strstr(path, "glob://") != path) {
740:             sprintf(&path, 0, "glob://%s", path);
741:         }
742:         IF (path)
743:             spl_filesystem_dir_open(intern, path);
744:         efree(path);
745:         } else
746:             IF (defined(PHP_WIN32))
747:                 spl_filesystem_dir_open(intern, path);
748:             else
749:                 spl_filesystem_dir_open(intern, path);

```



```

750: }
751: /* }}} */
752:
753: /* {{{ proto void DirectoryIterator::__construct (string path)
754:  * Constructs a new dir iterator from a path. */
755: SPL_METHOD(DirectoryIterator, __construct)
756: {
757:     spl_filesystem_object_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, 0);
758: }
759: /* }}} */
760:
761: /* {{{ proto void DirectoryIterator::rewind()
762:  * Rewind dir back to the start */
763: SPL_METHOD(DirectoryIterator, rewind)
764: {
765:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
766: }
767: IF (zend_parse_parameters_none() == FAILURE) {
768:     return;
769: }
770:
771: intern->u.dir.index = 0;
772: IF (intern->u.dir.dirp) {
773:     php_stream_rewinddir (intern->u.dir.dirp);
774: }
775: spl_filesystem_dir_read (intern);
776: }
777: /* }}} */
778:
779: /* {{{ proto string DirectoryIterator::key()
780:  * Return current dir entry */
781: SPL_METHOD(DirectoryIterator, key)
782: {
783:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
784: }
785: IF (zend_parse_parameters_none() == FAILURE) {
786:     return;
787: }
788:
789: IF (intern->u.dir.dirp) {
790:     RETURN_LONG (intern->u.dir.index);
791: } else {
792:     RETURN_FALSE;
793: }
794: }
795: /* }}} */
796:
797: /* {{{ proto DirectoryIterator DirectoryIterator::current()
798:  * Return this (needed for Iterator interface) */
799: SPL_METHOD(DirectoryIterator, current)
800: {
801:     IF (zend_parse_parameters_none() == FAILURE) {
802:         return;
803:     }
804:     ZVAL_OBJ(&return_value, _Z_OBJ_P(getThis()));
805:     _Z_ADDREF_P(return_value);
806: }
807: /* }}} */
808:
809: /* {{{ proto void DirectoryIterator::next()
810:  * Move to next entry */
811: SPL_METHOD(DirectoryIterator, next)
812: {
813:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
814:     int skip_dots = SPL_HAS_FLAG (intern->flags, SPL_FILE_DIR_SKIPDOTS);
815:     IF (zend_parse_parameters_none() == FAILURE) {
816:         return;
817:     }
818: }
819:
820: intern->u.dir.index++;
821: do {
822:     spl_filesystem_dir_read (intern);
823: } while (skip_dots && spl_filesystem_is_dot (intern->u.dir.entry_d_name));
824: IF (intern->u.dir.name) {
825:     /*new (intern->u.dir.name);
826:     intern->u.dir.name = NULL;
827: }
828: }
829: /* }}} */
830:
831: /* {{{ proto void DirectoryIterator::seek (int position)
832:  * Seek to the given position */
833: SPL_METHOD(DirectoryIterator, seek)
834: {
835:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
836:     zval retval;
837:     zend_long pos;
838:     IF (zend_parse_parameters (ZEND_NUM_ARGS(), "l", &pos) == FAILURE) {
839:         return;
840:     }
841: }
842:
843: IF (intern->u.dir.index > pos) {
844:     /* we first rewind */
845:     zend_call_method_with_0_params (&EX(This), _Z_OBJCE (EX(This)), &intern->u.dir.func_rewind, "rewind", NULL);
846: }
847:
848: while (intern->u.dir.index < pos) {
849:     int valid = 0;
850:     zend_call_method_with_0_params (&EX(This), _Z_OBJCE (EX(This)), &intern->u.dir.func_valid, "valid", &retval);
851:     if (!IS_BOOL(retval)) {
852:         valid = zend_is_true(retval);
853:     }
854:     if (valid) {
855:         zend_throw_exception_ex (spl_ce_OutOfBoundsException, 0, "Seek position '%ZEND_LONG_FMT%' is out of range", pos);
856:     }
857:     return;
858: }
859: zend_call_method_with_0_params (&EX(This), _Z_OBJCE (EX(This)), &intern->u.dir.func_next, "next", NULL);
860: }
861: /* }}} */
862:
863: /* {{{ proto string DirectoryIterator::valid()
864:  * Check whether dir contains more entries */
865: SPL_METHOD(DirectoryIterator, valid)
866: {
867:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
868: }
869: IF (zend_parse_parameters_none() == FAILURE) {
870:     return;
871: }
872:
873: RETURN_BOOL (intern->u.dir.entry_d_name[0] != '\0');
874: }
875: /* }}} */
876:
877: /* {{{ proto string SplFileInfo::getPath()
878:  * Return the path */
879: SPL_METHOD(SplFileInfo, getPath)
880: {
881:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
882:     char *path;
883:     size_t path_len;
884:     IF (zend_parse_parameters_none() == FAILURE) {
885:         return;
886:     }
887: }
888:
889: path = spl_filesystem_object_get_path (intern, &path_len);
890: RETURN_STRING (path, path_len);
891: }
892: /* }}} */
893:
894: /* {{{ proto string SplFileInfo::getFilename()
895:  * Return filename only */
896: SPL_METHOD(SplFileInfo, getFilename)
897: {
898:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
899:     size_t path_len;
900:     IF (zend_parse_parameters_none() == FAILURE) {
901:         return;
902:     }
903: }
904:
905: spl_filesystem_object_get_path (intern, &path_len);
906:
907: IF (path_len < path_len < intern->u.dir.name_len) {
908:     RETURN_STRING (intern->u.dir.name + path_len + 1, intern->u.dir.name_len - (path_len + 1));
909: } else {
910:     RETURN_STRING (intern->u.dir.name, intern->u.dir.name_len);
911: }
912: }
913: /* }}} */
914:
915: /* {{{ proto string DirectoryIterator::getFilename()
916:  * Return filename of current dir entry */
917: SPL_METHOD(DirectoryIterator, getFilename)
918: {
919:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
920: }
921: IF (zend_parse_parameters_none() == FAILURE) {
922:     return;
923: }
924:
925: RETURN_STRING (intern->u.dir.entry_d_name);
926: }
927: /* }}} */
928:
929: /* {{{ proto string SplFileInfo::getExtension()
930:  * Return file extension component of path */
931: SPL_METHOD(SplFileInfo, getExtension)
932: {
933:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
934:     char *fname = NULL;
935:     const char *p;
936:     size_t flen;
937:     size_t path_len;

```

```

938:     size_t idx;
939:     zend_string *ret;
940:     IF (zend_parse_parameters_none() == FAILURE) {
941:         return;
942:     }
943: }
944:
945: spl_filesystem_object_get_path (intern, &path_len);
946:
947: IF (path_len < path_len < intern->u.dir.name_len) {
948:     fname = intern->u.dir.name + path_len + 1;
949:     flen = intern->u.dir.name_len - (path_len + 1);
950: } else {
951:     fname = intern->u.dir.name;
952:     flen = intern->u.dir.name_len;
953: }
954:
955: ret = php_basename (fname, flen, NULL, 0);
956:
957: p = zend_search (ZSTR_VAL (ret), '.', ZSTR_LEN (ret));
958: IF (p) {
959:     idx = p - ZSTR_VAL (ret);
960:     RETVAL_STRING (ZSTR_VAL (ret) + idx + 1, ZSTR_LEN (ret) - idx - 1);
961:     zend_string_release (ret);
962:     return;
963: } else {
964:     zend_string_release (ret);
965:     RETURN_EMPTY_STRING ();
966: }
967: }
968: /* }}} */
969:
970: /* {{{ proto string DirectoryIterator::getExtension()
971:  * Return the file extension component of path */
972: SPL_METHOD(DirectoryIterator, getExtension)
973: {
974:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
975:     const char *p;
976:     size_t idx;
977:     zend_string *fname;
978:     IF (zend_parse_parameters_none() == FAILURE) {
979:         return;
980:     }
981: }
982:
983: fname = php_basename (intern->u.dir.entry_d_name, strlen (intern->u.dir.entry_d_name), NULL, 0);
984:
985: p = zend_search (ZSTR_VAL (fname), '.', ZSTR_LEN (fname));
986: IF (p) {
987:     idx = p - ZSTR_VAL (fname);
988:     RETVAL_STRING (ZSTR_VAL (fname) + idx + 1, ZSTR_LEN (fname) - idx - 1);
989:     zend_string_release (fname);
990:     return;
991: } else {
992:     zend_string_release (fname);
993:     RETURN_EMPTY_STRING ();
994: }
995: /* }}} */
996:
997: /* {{{ proto string SplFileInfo::getBasename (string suffix)
998:  * Return filename component of path */
999: SPL_METHOD(SplFileInfo, getBasename)
1000: {
1001:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
1002:     char *fname, *suffix = 0;
1003:     size_t flen;
1004:     size_t slen = 0, path_len;
1005:     IF (zend_parse_parameters (ZEND_NUM_ARGS(), "s", &suffix, &slen) == FAILURE) {
1006:         return;
1007:     }
1008: }
1009:
1010: spl_filesystem_object_get_path (intern, &path_len);
1011:
1012: IF (path_len < path_len < intern->u.dir.name_len) {
1013:     fname = intern->u.dir.name + path_len + 1;
1014:     flen = intern->u.dir.name_len - (path_len + 1);
1015: } else {
1016:     fname = intern->u.dir.name;
1017:     flen = intern->u.dir.name_len;
1018: }
1019:
1020: RETURN_STR (php_basename (fname, flen, suffix, slen));
1021: }
1022: /* }}} */
1023:
1024: /* {{{ proto string DirectoryIterator::getBasename (string suffix)
1025:  * Return filename component of current dir entry */
1026: SPL_METHOD(DirectoryIterator, getBasename)
1027: {
1028:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
1029:     char *suffix = 0;
1030:     size_t slen = 0;
1031:     zend_string *fname;
1032:     IF (zend_parse_parameters (ZEND_NUM_ARGS(), "s", &suffix, &slen) == FAILURE) {
1033:         return;
1034:     }
1035: }
1036:
1037: fname = php_basename (intern->u.dir.entry_d_name, strlen (intern->u.dir.entry_d_name), suffix, slen);
1038:
1039: RETVAL_STR (fname);
1040: }
1041: /* }}} */
1042:
1043: /* {{{ proto string SplFileInfo::getPathname()
1044:  * Return path and filename */
1045: SPL_METHOD(SplFileInfo, getPathname)
1046: {
1047:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
1048:     char *path;
1049:     size_t path_len;
1050:     IF (zend_parse_parameters_none() == FAILURE) {
1051:         return;
1052:     }
1053: }
1054:
1055: path = spl_filesystem_object_get_pathname (intern, &path_len);
1056: IF (path != NULL) {
1057:     RETURN_STRING (path, path_len);
1058: } else {
1059:     RETURN_FALSE;
1060: }
1061: /* }}} */
1062:
1063: /* {{{ proto string FilesystemIterator::key()
1064:  * Return getFilename() or getFileInfo() depending on flags */
1065: SPL_METHOD(FilesystemIterator, key)
1066: {
1067:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
1068: }
1069: IF (zend_parse_parameters_none() == FAILURE) {
1070:     return;
1071: }
1072:
1073: IF (SPL_FILE_DIR_CURRENT (intern, SPL_FILE_DIR_CURRENT_AS_PATHNAME)) {
1074:     RETURN_STRING (intern->u.dir.entry_d_name);
1075: } else {
1076:     spl_filesystem_object_get_file_name (intern);
1077:     RETURN_STRING (intern->u.dir.name, intern->u.dir.name_len);
1078: }
1079: }
1080: /* }}} */
1081:
1082: /* {{{ proto string FilesystemIterator::current()
1083:  * Return getFilename() or getFileInfo() depending on flags */
1084: SPL_METHOD(FilesystemIterator, current)
1085: {
1086:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
1087: }
1088: IF (zend_parse_parameters_none() == FAILURE) {
1089:     return;
1090: }
1091:
1092: IF (SPL_FILE_DIR_CURRENT (intern, SPL_FILE_DIR_CURRENT_AS_PATHNAME)) {
1093:     spl_filesystem_object_get_file_name (intern);
1094:     RETURN_STRING (intern->u.dir.name, intern->u.dir.name_len);
1095: } else IF (SPL_FILE_DIR_CURRENT (intern, SPL_FILE_DIR_CURRENT_AS_FILEINFO)) {
1096:     spl_filesystem_object_get_file_name (intern);
1097:     spl_filesystem_object_get_cwata_type (0, intern, SPL_FS_INFO, NULL, return_value);
1098: } else {
1099:     ZVAL_OBJ (&return_value, _Z_OBJ_P (getThis()));
1100:     _Z_ADDREF_P (return_value);
1101:     /*RETURN_STRING (intern->u.dir.entry_d_name, 1);*/
1102: }
1103: }
1104: /* }}} */
1105:
1106: /* {{{ proto bool DirectoryIterator::isDot()
1107:  * Return true if current entry is '.' or './' */
1108: SPL_METHOD(DirectoryIterator, isDot)
1109: {
1110:     spl_filesystem_object_construct ('intern = _Z_SPFILESYSTEM_P(getThis());
1111: }
1112: IF (zend_parse_parameters_none() == FAILURE) {
1113:     return;
1114: }
1115:
1116: RETURN_BOOL (spl_filesystem_is_dot (intern->u.dir.entry_d_name));
1117: }
1118: /* }}} */
1119:
1120: /* {{{ proto void SplFileInfo::__construct (string filename)
1121:  * Constructs a new SplFileInfo from a path */
1122: /* When the constructor gets called the object is already created
1123:  * by the engine, so we must only call 'additional' initializations.
1124:  */
1125: SPL_METHOD(SplFileInfo, __construct)

```

```

1126: {
1127:     spl_filesystem_object *intern;
1128:     char *path;
1129:     size_t len;
1130:
1131:     if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "s", &path, &len) == FAILURE) {
1132:         return;
1133:     }
1134:
1135:     intern = _spl_filesystem_P(getThis());
1136:
1137:     spl_filesystem_info_set_filename(intern, path, len, 1);
1138:
1139:     /* intern->type = SPL_FS_INFO already set */
1140:
1141:     /* {{{ */
1142:
1143:     /* {{{ FileinfoFunction */
1144:     FileinfoFunction(func_name, func_num) {
1145:         SPL_METHOD(splFileInfo, func_name) {
1146:             \
1147:             spl_filesystem_object *intern = _spl_filesystem_P(getThis()); \
1148:             zend_error_handling error_handling; \
1149:             if (zend_parse_parameters_none() == FAILURE) { \
1150:                 return; \
1151:             } \
1152:             \
1153:             zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling); \
1154:             spl_filesystem_object_get_file_name(intern); \
1155:             php_stat(intern->file_name, intern->file_name_len, func_num, return_value); \
1156:             zend_restore_error_handling(&error_handling); \
1157:             \
1158:             /* }}} */
1159:
1160:     /* {{{ proto int splFileInfo:getPerms()
1161:      * Get file permissions */
1162:     FileinfoFunction(getPerms, FS_PERMS) {
1163:         /* }}} */
1164:
1165:     /* {{{ proto int splFileInfo:getInode()
1166:      * Get file inode */
1167:     FileinfoFunction(getInode, FS_INODE)
1168:     /* }}} */
1169:
1170:     /* {{{ proto int splFileInfo:getSize()
1171:      * Get file size */
1172:     FileinfoFunction(getSize, FS_SIZE)
1173:     /* }}} */
1174:
1175:     /* {{{ proto int splFileInfo:getOwner()
1176:      * Get file owner */
1177:     FileinfoFunction(getOwner, FS_OWNER)
1178:     /* }}} */
1179:
1180:     /* {{{ proto int splFileInfo:getGroup()
1181:      * Get file group */
1182:     FileinfoFunction(getGroup, FS_GROUP)
1183:     /* }}} */
1184:
1185:     /* {{{ proto int splFileInfo:getAtime()
1186:      * Get last access time of file */
1187:     FileinfoFunction(getAtime, FS_ATIME)
1188:     /* }}} */
1189:
1190:     /* {{{ proto int splFileInfo:getMTime()
1191:      * Get last modification time of file */
1192:     FileinfoFunction(getMTime, FS_MTIME)
1193:     /* }}} */
1194:
1195:     /* {{{ proto int splFileInfo:getCtime()
1196:      * Get inode modification time of file */
1197:     FileinfoFunction(getCtime, FS_CTIME)
1198:     /* }}} */
1199:
1200:     /* {{{ proto string splFileInfo:getType()
1201:      * Get file type */
1202:     FileinfoFunction(getType, FS_TYPE)
1203:     /* }}} */
1204:
1205:     /* {{{ proto bool splFileInfo:isWritable()
1206:      * Returns true if file can be written */
1207:     FileinfoFunction(isWritable, FS_IS_W)
1208:     /* }}} */
1209:
1210:     /* {{{ proto bool splFileInfo:isReadable()
1211:      * Returns true if file can be read */
1212:     FileinfoFunction(isReadable, FS_IS_R)
1213:     /* }}} */
1214:
1215:     /* {{{ proto bool splFileInfo:isExecutable()
1216:      * Returns true if file is executable */
1217:     FileinfoFunction(isExecutable, FS_IS_X)
1218:     /* }}} */
1219:
1220:     /* {{{ proto bool splFileInfo:isFile()
1221:      * Returns true if file is a regular file */
1222:     FileinfoFunction(isFile, FS_IS_FILE)
1223:     /* }}} */
1224:
1225:     /* {{{ proto bool splFileInfo:isDir()
1226:      * Returns true if file is directory */
1227:     FileinfoFunction(isDir, FS_IS_DIR)
1228:     /* }}} */
1229:
1230:     /* {{{ proto bool splFileInfo:isLink()
1231:      * Returns true if file is symbolic link */
1232:     FileinfoFunction(isLink, FS_IS_LINK)
1233:     /* }}} */
1234:
1235:     /* {{{ proto string splFileInfo:getLinkTarget()
1236:      * Return the target of a symbolic link */
1237:     SPL_METHOD(splFileInfo, getLinkTarget)
1238:     /* }}} */
1239:
1240:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1241:     symlink &ret;
1242:     char buff[MAXPATHLEN];
1243:     zend_error_handling error_handling;
1244:
1245:     if (zend_parse_parameters_none() == FAILURE) {
1246:         return;
1247:     }
1248:
1249:     zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
1250:
1251:     if (defined(PHP_WIN32)) { HAVE_SYMLINK
1252:     if (intern->file_name == NULL) {
1253:         php_error_docref(NULL, E_WARNING, "Empty filename");
1254:         return;
1255:     }
1256:     if (strlen(intern->file_name) > MAXPATHLEN) {
1257:         php_error_docref(NULL, E_WARNING, "No such file or directory");
1258:         return;
1259:     }
1260:     ret = php_spl_readlink(expanded_path, buff, MAXPATHLEN - 1);
1261:     } else {
1262:     ret = php_spl_readlink(intern->file_name, buff, MAXPATHLEN - 1);
1263:     }
1264:     return;
1265: }
1266:
1267: if (ret == -1) /* always fail if not implemented */
1268:
1269: #endif
1270:
1271: if (ret == -1) {
1272:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Unable to read link %s, error: %s", intern->file_name, strerror(errno));
1273:     return;
1274: } else {
1275:     /* Append NULL to the end of the string */
1276:     buff[ret] = '\0';
1277:
1278:     RETVAL_STRING(buff, ret);
1279: }
1280:
1281: zend_restore_error_handling(&error_handling);
1282: /* }}} */
1283:
1284: if (HAVE_REALPATH) {
1285:     /* {{{ proto string splFileInfo:getRealPath()
1286:      * Return the resolved path */
1287:     SPL_METHOD(splFileInfo, getRealPath)
1288:     /* }}} */
1289:
1290:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1291:     char buff[MAXPATHLEN];
1292:     char *filename;
1293:     zend_error_handling error_handling;
1294:
1295:     if (zend_parse_parameters_none() == FAILURE) {
1296:         return;
1297:     }
1298:
1299:     zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
1300:
1301:     if (intern->type == SPL_FS_DIR && !intern->file_name && !intern->u.dir.entry_dname[0]) {
1302:         spl_filesystem_object_get_file_name(intern);
1303:     }
1304:
1305:     if (intern->orig_path) {
1306:         filename = intern->orig_path;
1307:     } else {
1308:         filename = intern->file_name;
1309:     }
1310:
1311:     if (filename && !VOWD_REALPATH(filename, buff)) {
1312:         if (VOWD_ACCESS(buff, F_OK)) {
1313:             RETVAL_FALSE;
1314:         } else
1315:
1316:
1317:
1318:
1319:
1320:
1321:
1322:
1323:
1324: #endif
1325:
1326: RETVAL_STRING(buff);
1327:
1328: } else {
1329:     RETVAL_FALSE;
1330: }
1331:
1332: }
1333:
1334: }
1335:
1336: }
1337:
1338: }
1339:
1340: }
1341:
1342: }
1343:
1344: }
1345:
1346: }
1347:
1348: }
1349:
1350: }
1351:
1352: }
1353:
1354: }
1355:
1356: }
1357:
1358: }
1359:
1360: }
1361:
1362: }
1363:
1364: }
1365:
1366: }
1367:
1368: }
1369:
1370: }
1371:
1372: }
1373:
1374: }
1375:
1376: }
1377:
1378: }
1379:
1380: }
1381:
1382: }
1383:
1384: }
1385:
1386: }
1387:
1388: }
1389:
1390: }
1391:
1392: }
1393:
1394: }
1395:
1396: }
1397:
1398: }
1399:
1400: }
1401:
1402: }
1403:
1404: }
1405:
1406: }
1407:
1408: }
1409:
1410: }
1411:
1412: }
1413:
1414: }
1415:
1416: }
1417:
1418: }
1419:
1420: }
1421:
1422: }
1423:
1424: }
1425:
1426: }
1427:
1428: }
1429:
1430: }
1431:
1432: }
1433:
1434: }
1435:
1436: }
1437:
1438: }
1439:
1440: }
1441:
1442: }
1443:
1444: }
1445:
1446: }
1447:
1448: }
1449:
1450: }
1451:
1452: }
1453:
1454: }
1455:
1456: }
1457:
1458: }
1459:
1460: }
1461:
1462: }
1463:
1464: }
1465:
1466: }
1467:
1468: }
1469:
1470: }
1471:
1472: }
1473:
1474: }
1475:
1476: }
1477:
1478: }
1479:
1480: }
1481:
1482: }
1483:
1484: }
1485:
1486: }
1487:
1488: }
1489:
1490: }
1491:
1492: }
1493:
1494: }
1495:
1496: }
1497:
1498: }
1499:
1500: }
1501:
1502: }
1503:
1504: }
1505:
1506: }
1507:
1508: }
1509:
1510: }
1511:
1512: }
1513:
1514: }
1515:
1516: }
1517:
1518: }
1519:
1520: }
1521:
1522: }
1523:
1524: }
1525:
1526: }
1527:
1528: }
1529:
1530: }
1531:
1532: }
1533:
1534: }
1535:
1536: }
1537:
1538: }
1539:
1540: }
1541:
1542: }
1543:
1544: }
1545:
1546: }
1547:
1548: }
1549:
1550: }
1551:
1552: }
1553:
1554: }
1555:
1556: }
1557:
1558: }
1559:
1560: }
1561:
1562: }
1563:
1564: }
1565:
1566: }
1567:
1568: }
1569:
1570: }
1571:
1572: }
1573:
1574: }
1575:
1576: }
1577:
1578: }
1579:
1580: }
1581:
1582: }
1583:
1584: }
1585:
1586: }
1587:
1588: }
1589:
1590: }
1591:
1592: }
1593:
1594: }
1595:
1596: }
1597:
1598: }
1599:
1600: }
1601:
1602: }
1603:
1604: }
1605:
1606: }
1607:
1608: }
1609:
1610: }
1611:
1612: }
1613:
1614: }
1615:
1616: }
1617:
1618: }
1619:
1620: }
1621:
1622: }
1623:
1624: }
1625:
1626: }
1627:
1628: }
1629:
1630: }
1631:
1632: }
1633:
1634: }
1635:
1636: }
1637:
1638: }
1639:
1640: }
1641:
1642: }
1643:
1644: }
1645:
1646: }
1647:
1648: }
1649:
1650: }
1651:
1652: }
1653:
1654: }
1655:
1656: }
1657:
1658: }
1659:
1660: }
1661:
1662: }
1663:
1664: }
1665:
1666: }
1667:
1668: }
1669:
1670: }
1671:
1672: }
1673:
1674: }
1675:
1676: }
1677:
1678: }
1679:
1680: }
1681:
1682: }
1683:
1684: }
1685:
1686: }
1687:
1688: }
1689:
1690: }
1691:
1692: }
1693:
1694: }
1695:
1696: }
1697:
1698: }
1699:
1700: }
1701:
1702: }
1703:
1704: }
1705:
1706: }
1707:
1708: }
1709:
1710: }
1711:
1712: }
1713:
1714: }
1715:
1716: }
1717:
1718: }
1719:
1720: }
1721:
1722: }
1723:
1724: }
1725:
1726: }
1727:
1728: }
1729:
1730: }
1731:
1732: }
1733:
1734: }
1735:
1736: }
1737:
1738: }
1739:
1740: }
1741:
1742: }
1743:
1744: }
1745:
1746: }
1747:
1748: }
1749:
1750: }
1751:
1752: }
1753:
1754: }
1755:
1756: }
1757:
1758: }
1759:
1760: }
1761:
1762: }
1763:
1764: }
1765:
1766: }
1767:
1768: }
1769:
1770: }
1771:
1772: }
1773:
1774: }
1775:
1776: }
1777:
1778: }
1779:
1780: }
1781:
1782: }
1783:
1784: }
1785:
1786: }
1787:
1788: }
1789:
1790: }
1791:
1792: }
1793:
1794: }
1795:
1796: }
1797:
1798: }
1799:
1800: }
1801:
1802: }
1803:
1804: }
1805:
1806: }
1807:
1808: }
1809:
1810: }
1811:
1812: }
1813:
1814: }
1815:
1816: }
1817:
1818: }
1819:
1820: }
1821:
1822: }
1823:
1824: }
1825:
1826: }
1827:
1828: }
1829:
1830: }
1831:
1832: }
1833:
1834: }
1835:
1836: }
1837:
1838: }
1839:
1840: }
1841:
1842: }
1843:
1844: }
1845:
1846: }
1847:
1848: }
1849:
1850: }
1851:
1852: }
1853:
1854: }
1855:
1856: }
1857:
1858: }
1859:
1860: }
1861:
1862: }
1863:
1864: }
1865:
1866: }
1867:
1868: }
1869:
1870: }
1871:
1872: }
1873:
1874: }
1875:
1876: }
1877:
1878: }
1879:
1880: }
1881:
1882: }
1883:
1884: }
1885:
1886: }
1887:
1888: }
1889:
1890: }
1891:
1892: }
1893:
1894: }
1895:
1896: }
1897:
1898: }
1899:
1900: }
1901:
1902: }
1903:
1904: }
1905:
1906: }
1907:
1908: }
1909:
1910: }
1911:
1912: }
1913:
1914: }
1915:
1916: }
1917:
1918: }
1919:
1920: }
1921:
1922: }
1923:
1924: }
1925:
1926: }
1927:
1928: }
1929:
1930: }
1931:
1932: }
1933:
1934: }
1935:
1936: }
1937:
1938: }
1939:
1940: }
1941:
1942: }
1943:
1944: }
1945:
1946: }
1947:
1948: }
1949:
1950: }
1951:
1952: }
1953:
1954: }
1955:
1956: }
1957:
1958: }
1959:
1960: }
1961:
1962: }
1963:
1964: }
1965:
1966: }
1967:
1968: }
1969:
1970: }
1971:
1972: }
1973:
1974: }
1975:
1976: }
1977:
1978: }
1979:
1980: }
1981:
1982: }
1983:
1984: }
1985:
1986: }
1987:
1988: }
1989:
1990: }
1991:
1992: }
1993:
1994: }
1995:
1996: }
1997:
1998: }
1999:
2000: }
2001:
2002: }
2003:
2004: }
2005:
2006: }
2007:
2008: }
2009:
2010: }
2011:
2012: }
2013:
2014: }
2015:
2016: }
2017:
2018: }
2019:
2020: }
2021:
2022: }
2023:
2024: }
2025:
2026: }
2027:
2028: }
2029:
2030: }
2031:
2032: }
2033:
2034: }
2035:
2036: }
2037:
2038: }
2039:
2040: }
2041:
2042: }
2043:
2044: }
2045:
2046: }
2047:
2048: }
2049:
2050: }
2051:
2052: }
2053:
2054: }
2055:
2056: }
2057:
2058: }
2059:
2060: }
2061:
2062: }
2063:
2064: }
2065:
2066: }
2067:
2068: }
2069:
2070: }
2071:
2072: }
2073:
2074: }
2075:
2076: }
2077:
2078: }
2079:
2080: }
2081:
2082: }
2083:
2084: }
2085:
2086: }
2087:
2088: }
2089:
2090: }
2091:
2092: }
2093:
2094: }
2095:
2096: }
2097:
2098: }
2099:
2100: }
2101:
2102: }
2103:
2104: }
2105:
2106: }
2107:
2108: }
2109:
2110: }
2111:
2112: }
2113:
2114: }
2115:
2116: }
2117:
2118: }
2119:
2120: }
2121:
2122: }
2123:
2124: }
2125:
2126: }
2127:
2128: }
2129:
2130: }
2131:
2132: }
2133:
2134: }
2135:
2136: }
2137:
2138: }
2139:
2140: }
2141:
2142: }
2143:
2144: }
2145:
2146: }
2147:
2148: }
2149:
2150: }
2151:
2152: }
2153:
2154: }
2155:
2156: }
2157:
2158: }
2159:
2160: }
2161:
2162: }
2163:
2164: }
2165:
2166: }
2167:
2168: }
2169:
2170: }
2171:
2172: }
2173:
2174: }
2175:
2176: }
2177:
2178: }
2179:
2180: }
2181:
2182: }
2183:
2184: }
2185:
2186: }
2187:
2188: }
2189:
2190: }
2191:
2192: }
2193:
2194: }
2195:
2196: }
2197:
2198: }
2199:
2200: }
2201:
2202: }
2203:
2204: }
2205:
2206: }
2207:
2208: }
2209:
2210: }
2211:
2212: }
2213:
2214: }
2215:
2216: }
2217:
2218: }
2219:
2220: }
2221:
2222: }
2223:
2224: }
2225:
2226: }
2227:
2228: }
2229:
2230: }
2231:
2232: }
2233:
2234: }
2235:
2236: }
2237:
2238: }
2239:
2240: }
2241:
2242: }
2243:
2244: }
2245:
2246: }
2247:
2248: }
2249:
2250: }
2251:
2252: }
2253:
2254: }
2255:
2256: }
2257:
2258: }
2259:
2260: }
2261:
2262: }
2263:
2264: }
2265:
2266: }
2267:
2268: }
2269:
2270: }
2271:
2272: }
2273:
2274: }
2275:
2276: }
2277:
2278: }
2279:
2280: }
2281:
2282: }
2283:
2284: }
2285:
2286: }
2287:
2288: }
2289:
2290: }
2291:
2292: }
2293:
2294: }
2295:
2296: }
2297:
2298: }
2299:
2300: }
2301:
2302: }
2303:
2304: }
2305:
2306: }
2307:
2308: }
2309:
2310: }
2311:
2312: }
2313:
2314: }
2315:
2316: }
2317:
2318: }
2319:
2320: }
2321:
2322: }
2323:
2324: }
2325:
2326: }
2327:
2328: }
2329:
2330: }
2331:
2332: }
2333:
2334: }
2335:
2336: }
2337:
2338: }
2339:
2340: }
2341:
2342: }
2343:
2344: }
2345:
2346: }
2347:
2348: }
2349:
2350: }
2351:
2352: }
2353:
2354: }
2355:
2356: }
2357:
2358: }
2359:
2360: }
2361:
2362: }
2363:
2364: }
2365:
2366: }
2367:
2368: }
2369:
2370: }
2371:
2372: }
2373:
2374: }
2375:
2376: }
2377:
2378: }
2379:
2380: }
2381:
2382: }
2383:
2384: }
2385:
2386: }
2387:
2388: }
2389:
2390: }
2391:
2392: }
2393:
2394: }
2395:
2396: }
2397:
2398: }
2399:
2400: }
2401:
2402: }
2403:
2404: }
2405:
2406: }
2407:
2408: }
2409:
2410: }
2411:
2412: }
2413:
2414: }
2415:
2416: }
2417:
2418: }
2419:
2420: }
2421:
2422: }
2423:
2424: }
2425:
2426: }
2427:
2428: }
2429:
2430: }
2431:
2432: }
2433:
2434: }
2435:
2436: }
2437:
2438: }
2439:
2440: }
2441:
2442: }
2443:
2444: }
2445:
2446: }
2447:
2448: }
2449:
2450: }
2451:
2452: }
2453:
2454: }
2455:
2456: }
2457:
2458: }
2459:
2460: }
2461:
2462: }
2463:
2464: }
2465:
2466: }
2467:
2468: }
2469:
2470: }
2471:
2472: }
2473:
2474: }
2475:
2476: }
2477:
2478: }
2479:
2480: }
2481:
2482: }
2483:
2484: }
2485:
2486: }
2487:
2488: }
2489:
2490: }
2491:
2492: }
2493:
2494: }
2495:
2496: }
2497:
2498: }
2499:
2500: }
2501:
2502: }
2503:
2504: }
2505:
2506: }
2507:
2508: }
2509:
2510: }
2511:
2512: }
2513:
2514: }
2515:
2516: }
2517:
2518: }
2519:
2520: }
2521:
2522: }
2523:
2524: }
2525:
2526: }
2527:
2528: }
2529:
2530: }
2531:
2532: }
2533:
2534: }
2535:
2536: }
2537:
2538: }
2539:
2540: }
2541:
2542: }
2543:
2544: }
2545:
2546: }
2547:
2548: }
2549:
2550: }
2551:
2552: }
2553:
2554: }
2555:
2556: }
2557:
2558: }
2559:
2560: }
2561:
2562: }
2563:
2564: }
2565:
2566: }
2567:
2568: }
2569:
2570: }
2571:
2572: }
2573:
2574: }
2575:
2576: }
2577:
2578: }
2579:
2580: }
2581:
2582: }
2583:
2584: }
2585:
2586: }
2587:
2588: }
2589:
2590: }
2591:
2592: }
2593:
2594: }
2595:
2596: }
2597:
2598: }
2599:
2600: }
2601:
2602: }
2603:
2604: }
2605:
2606: }
2607:
2608: }
2609:
2610: }
2611:
2612: }
2613:
2614: }
2615:
2616: }
2617:
2618: }
2619:
2620: }
2621:
2622: }
2623:
2624: }
2625:
2626: }
2627:
2628: }
2629:
2630: }
2631:
2632: }
2633:
2634: }
2635:
2636: }
2637:
2638: }
2639:
2640: }
2641:
2642: }
2643:
2644: }
2645:
2646: }
2647:
2648: }
2649:
2650: }
2651:
2652: }
2653:
2654: }
2655:
2656: }
2657:
2658: }
2659:
2660: }
2661:
2662: }
2663:
2664: }
2665:
2666: }
2667:
2668: }
2669:
2670: }
2671:
2672: }
2673:
2674: }
2675:
2676: }
2677:
2678: }
2679:
2680: }
2681:
2682: }
2683:
2684: }
2685:
2686: }
2687:
2688: }
2689:
2690: }
2691:
2692: }
2693:
2694: }
2695:
2696: }
2697:
2698: }
2699:
2700: }
2701:
2702: }
2703:
2704: }
2705:
2706: }
2707:
2708: }
2709:
2710: }
2711:
2712: }
2713:
2714: }
2715:
2716: }
2717:
2718: }
2719:
2720: }
2721:
2722: }
2723:
2724: }
2725:
2726: }
2727:
2728: }
2729:
2730: }
2731:
2732: }
2733:
2734: }
2735:
2736: }
2737:
2738: }
2739:
2740: }
2741:
2742: }
2743:
2744: }
2745:
2746: }
2747:
2748: }
2749:
2750: }
2751:
2752: }
2753:
2754: }
2755:
2756: }
2757:
2758: }
2759:
2760: }
2761:
2762: }
2763:
2764: }
2765:
2766: }
2767:
2768: }
2769:
2770: }
2771:
2772: }
2773:
2774: }
2775:
2776: }
2777:
2778: }
2779:
2780: }
2781:
2782: }
2783:
2784: }
2785:
2786: }
2787:
2788: }
2789:
2790: }
2791:
2792: }
2793:
2794: }
2795:
2796: }
2797:
2798: }
2799:
2800: }
2801:
2802: }
2803:
2804: }
2805:
2806: }
2807:
2808: }
2809:
2810: }
2811:
2812: }
2813:
2814: }
2815:
2816
```

```

1502: }
1503: /* }}} */
1504:
1505: /* {{{ proto RecursiveDirectoryIterator DirectoryIterator::getChildren()
1506:  * Returns an iterator for the current entry if it is a directory */
1507: SPL_METHOD(RecursiveDirectoryIterator, getChildren)
1508: {
1509:     zval $path, $flags;
1510:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
1511:     spl_filesystem_object *subdir;
1512:     char slash = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_UNIXPATHS) ? '/' : DEFAULT_SLASH;
1513:
1514:     IF (zend_parse_parameters_none() == FAILURE) {
1515:         return;
1516:     }
1517:
1518:     spl_filesystem_object_get_file_name(intern);
1519:
1520:     ZVAL_LONG($flags, intern->flags);
1521:     ZVAL_STRING($path, intern->file_name, intern->file_name_len);
1522:     spl_instance_init_arg_ext(2, OBJCE_P(getThis()), return_value, $path, $flags);
1523:     zval_ptr_dtor($path);
1524:     zval_ptr_dtor($flags);
1525:
1526:     subdir = _SPL_FILESYSTEM_P(return_value);
1527:     IF (subdir) {
1528:         IF (intern->u.dir.sub_path & intern->u.dir.sub_path[0]) {
1529:             subdir->u.dir.sub_path_len = sprintf(&subdir->u.dir.sub_path, 0, "%s%s", intern->u.dir.sub_path, slash, intern->u.dir.entry_d_name);
1530:         } else {
1531:             subdir->u.dir.sub_path_len = strlen(intern->u.dir.entry_d_name);
1532:             subdir->u.dir.sub_path = estrndup(intern->u.dir.entry_d_name, subdir->u.dir.sub_path_len);
1533:         }
1534:         subdir->info_class = intern->info_class;
1535:         subdir->file_class = intern->file_class;
1536:         subdir->oth = intern->oth;
1537:     }
1538: }
1539: /* }}} */
1540:
1541: /* {{{ proto void RecursiveDirectoryIterator::getSubPath()
1542:  * Get sub path */
1543: SPL_METHOD(RecursiveDirectoryIterator, getSubPath)
1544: {
1545:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
1546:
1547:     IF (zend_parse_parameters_none() == FAILURE) {
1548:         return;
1549:     }
1550:
1551:     IF (intern->u.dir.sub_path) {
1552:         RETURN_STRING(intern->u.dir.sub_path, intern->u.dir.sub_path_len);
1553:     } else {
1554:         RETURN_EMPTY_STRING();
1555:     }
1556: }
1557: /* }}} */
1558:
1559: /* {{{ proto void RecursiveDirectoryIterator::getSubPathname()
1560:  * Get sub path and file name */
1561: SPL_METHOD(RecursiveDirectoryIterator, getSubPathname)
1562: {
1563:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
1564:     char slash = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_UNIXPATHS) ? '/' : DEFAULT_SLASH;
1565:
1566:     IF (zend_parse_parameters_none() == FAILURE) {
1567:         return;
1568:     }
1569:
1570:     IF (intern->u.dir.sub_path) {
1571:         RETURN_NEW_STR(1, sprintf(0, "%s%s", intern->u.dir.sub_path, slash, intern->u.dir.entry_d_name));
1572:     } else {
1573:         RETURN_STRING(intern->u.dir.entry_d_name);
1574:     }
1575: }
1576: /* }}} */
1577:
1578: /* {{{ proto int RecursiveDirectoryIterator::__construct(string path [, int flags])
1579:  * Constructs a new dir iterator from a path. */
1580: SPL_METHOD(RecursiveDirectoryIterator, __construct)
1581: {
1582:     spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, DIT_CTOR_FLAGS);
1583: }
1584: /* }}} */
1585:
1586: #ifdef HAVE_GLOB
1587: /* {{{ proto int GlobIterator::__construct(string path [, int flags])
1588:  * Constructs a new dir iterator from a glob expression (no globlib needed). */
1589: SPL_METHOD(GlobIterator, __construct)
1590: {
1591:     spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, DIT_CTOR_FLAGS|DIT_CTOR_GLOB);
1592: }
1593: /* }}} */
1594:
1595: /* {{{ proto int GlobIterator::count()
1596:  * Return the number of directories and files found by globbing */
1597: SPL_METHOD(GlobIterator, count)
1598: {
1599:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
1600:
1601:     IF (zend_parse_parameters_none() == FAILURE) {
1602:         return;
1603:     }
1604:
1605:     IF (intern->u.dir.dirp && php_stream_is(intern->u.dir.dirp, &php_glob_stream_ops)) {
1606:         RETURN_LONG(php_glob_stream_get_count(intern->u.dir.dirp, NULL));
1607:     } else {
1608:         /* should not happen */
1609:         php_error_docref(NULL, E_ERROR, "GlobIterator lost glob state");
1610:     }
1611: }
1612: /* }}} */
1613: #endif /* HAVE_GLOB */
1614:
1615: /* {{{ forward declarations to the iterator handlers */
1616: static void spl_filesystem_dir_it_dtor(spl_filesystem_object *iter);
1617: static int spl_filesystem_dir_it_valid(spl_filesystem_object *iter);
1618: static zval *spl_filesystem_dir_it_current_data(spl_filesystem_object *iter);
1619: static void spl_filesystem_dir_it_current_key(spl_filesystem_object *iter, zval *key);
1620: static void spl_filesystem_dir_it_move_forward(spl_filesystem_object *iter);
1621: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter);
1622:
1623: /* iterator handler table */
1624: static const zend_object_iterator_funcs spl_filesystem_dir_it_funcs = {
1625:     spl_filesystem_dir_it_dtor,
1626:     spl_filesystem_dir_it_valid,
1627:     spl_filesystem_dir_it_current_data,
1628:     spl_filesystem_dir_it_current_key,
1629:     spl_filesystem_dir_it_move_forward,
1630:     spl_filesystem_dir_it_rewind,
1631:     NULL
1632: };
1633: /* }}} */
1634:
1635: /* {{{ spl_on_dir_get_iterator */
1636: zend_object_iterator *spl_filesystem_dir_get_iterator(zend_class_entry *ce, zval *object, int by_ref)
1637: {
1638:     spl_filesystem_iterator *iterator;
1639:     spl_filesystem_object *dir_object;
1640:
1641:     IF (by_ref) {
1642:         zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1643:         return NULL;
1644:     }
1645:
1646:     dir_object = _SPL_FILESYSTEM_P(object);
1647:     iterator = spl_filesystem_object_to_iterator(dir_object);
1648:     ZVAL_COPY(iterator->intern.data, object);
1649:     iterator->intern.funcs = spl_filesystem_dir_it_funcs;
1650:     /* ~current must be initialized! rewind doesn't set it and valid
1651:      * doesn't check whether it's set */
1652:     iterator->current = *object;
1653:
1654:     return iterator->intern;
1655: }
1656: /* }}} */
1657:
1658: /* {{{ spl_filesystem_dir_it_dtor */
1659: static void spl_filesystem_dir_it_dtor(spl_filesystem_object *iter)
1660: {
1661:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1662:
1663:     IF (!IS_ISNULL(iterator->intern.data)) {
1664:         zval *obj = iterator->intern.data;
1665:         zval_ptr_dtor(obj);
1666:     }
1667:
1668:     /* Otherwise we were called from the owning object free storage handler as
1669:      * it sets iterator->intern.data to IS_UNDEF.
1670:      * We don't even need to destroy iterator->current as we didn't add a
1671:      * reference to it in move_forward or get_iterator */
1672: }
1673: /* }}} */
1674:
1675: /* {{{ spl_filesystem_dir_it_valid */
1676: static int spl_filesystem_dir_it_valid(spl_filesystem_object *iter)
1677: {
1678:     spl_filesystem_object *object = spl_filesystem_iterator_to_object(spl_filesystem_iterator *)iter;
1679:
1680:     return object->u.dir.entry_d_name[0] != '\0' ? SUCCESS : FAILURE;
1681: }
1682: /* }}} */
1683:
1684: /* {{{ spl_filesystem_dir_it_current_data */
1685: static zval *spl_filesystem_dir_it_current_data(spl_filesystem_object *iter)
1686: {
1687:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1688:
1689:     return iterator->current;
1690: }
1691: /* }}} */
1692:
1693: /* {{{ spl_filesystem_dir_it_rewind */
1694: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1695: {
1696:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1697:
1698:     IF (!IS_ISNULL(iterator->intern.data)) {
1699:         zval *obj = iterator->intern.data;
1700:         zval_ptr_dtor(obj);
1701:     }
1702:
1703:     /* Otherwise we were called from the owning object free storage handler as
1704:      * it sets iterator->intern.data to IS_UNDEF.
1705:      * We don't even need to destroy iterator->current as we didn't add a
1706:      * reference to it in move_forward or get_iterator */
1707: }
1708: /* }}} */
1709:
1710: /* {{{ spl_filesystem_dir_it_move_forward */
1711: static void spl_filesystem_dir_it_move_forward(spl_filesystem_object *iter)
1712: {
1713:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1714:
1715:     IF (!IS_ISNULL(iterator->intern.data)) {
1716:         zval *obj = iterator->intern.data;
1717:         zval_ptr_dtor(obj);
1718:     }
1719:
1720:     /* Otherwise we were called from the owning object free storage handler as
1721:      * it sets iterator->intern.data to IS_UNDEF.
1722:      * We don't even need to destroy iterator->current as we didn't add a
1723:      * reference to it in move_forward or get_iterator */
1724: }
1725: /* }}} */
1726:
1727: /* {{{ spl_filesystem_dir_it_rewind */
1728: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1729: {
1730:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1731:
1732:     IF (!IS_ISNULL(iterator->intern.data)) {
1733:         zval *obj = iterator->intern.data;
1734:         zval_ptr_dtor(obj);
1735:     }
1736:
1737:     /* Otherwise we were called from the owning object free storage handler as
1738:      * it sets iterator->intern.data to IS_UNDEF.
1739:      * We don't even need to destroy iterator->current as we didn't add a
1740:      * reference to it in move_forward or get_iterator */
1741: }
1742: /* }}} */
1743:
1744: /* {{{ spl_filesystem_dir_it_rewind */
1745: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1746: {
1747:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1748:
1749:     IF (!IS_ISNULL(iterator->intern.data)) {
1750:         zval *obj = iterator->intern.data;
1751:         zval_ptr_dtor(obj);
1752:     }
1753:
1754:     /* Otherwise we were called from the owning object free storage handler as
1755:      * it sets iterator->intern.data to IS_UNDEF.
1756:      * We don't even need to destroy iterator->current as we didn't add a
1757:      * reference to it in move_forward or get_iterator */
1758: }
1759: /* }}} */
1760:
1761: /* {{{ spl_filesystem_dir_it_rewind */
1762: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1763: {
1764:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1765:
1766:     IF (!IS_ISNULL(iterator->intern.data)) {
1767:         zval *obj = iterator->intern.data;
1768:         zval_ptr_dtor(obj);
1769:     }
1770:
1771:     /* Otherwise we were called from the owning object free storage handler as
1772:      * it sets iterator->intern.data to IS_UNDEF.
1773:      * We don't even need to destroy iterator->current as we didn't add a
1774:      * reference to it in move_forward or get_iterator */
1775: }
1776: /* }}} */
1777:
1778: /* {{{ spl_filesystem_dir_it_rewind */
1779: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1780: {
1781:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1782:
1783:     IF (!IS_ISNULL(iterator->intern.data)) {
1784:         zval *obj = iterator->intern.data;
1785:         zval_ptr_dtor(obj);
1786:     }
1787:
1788:     /* Otherwise we were called from the owning object free storage handler as
1789:      * it sets iterator->intern.data to IS_UNDEF.
1790:      * We don't even need to destroy iterator->current as we didn't add a
1791:      * reference to it in move_forward or get_iterator */
1792: }
1793: /* }}} */
1794:
1795: /* {{{ spl_filesystem_dir_it_rewind */
1796: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1797: {
1798:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1799:
1800:     IF (!IS_ISNULL(iterator->intern.data)) {
1801:         zval *obj = iterator->intern.data;
1802:         zval_ptr_dtor(obj);
1803:     }
1804:
1805:     /* Otherwise we were called from the owning object free storage handler as
1806:      * it sets iterator->intern.data to IS_UNDEF.
1807:      * We don't even need to destroy iterator->current as we didn't add a
1808:      * reference to it in move_forward or get_iterator */
1809: }
1810: /* }}} */
1811:
1812: /* {{{ spl_filesystem_dir_it_rewind */
1813: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1814: {
1815:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1816:
1817:     IF (!IS_ISNULL(iterator->intern.data)) {
1818:         zval *obj = iterator->intern.data;
1819:         zval_ptr_dtor(obj);
1820:     }
1821:
1822:     /* Otherwise we were called from the owning object free storage handler as
1823:      * it sets iterator->intern.data to IS_UNDEF.
1824:      * We don't even need to destroy iterator->current as we didn't add a
1825:      * reference to it in move_forward or get_iterator */
1826: }
1827: /* }}} */
1828:
1829: /* {{{ spl_filesystem_dir_it_rewind */
1830: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1831: {
1832:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1833:
1834:     IF (!IS_ISNULL(iterator->intern.data)) {
1835:         zval *obj = iterator->intern.data;
1836:         zval_ptr_dtor(obj);
1837:     }
1838:
1839:     /* Otherwise we were called from the owning object free storage handler as
1840:      * it sets iterator->intern.data to IS_UNDEF.
1841:      * We don't even need to destroy iterator->current as we didn't add a
1842:      * reference to it in move_forward or get_iterator */
1843: }
1844: /* }}} */
1845:
1846: /* {{{ spl_filesystem_dir_it_rewind */
1847: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1848: {
1849:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1850:
1851:     IF (!IS_ISNULL(iterator->intern.data)) {
1852:         zval *obj = iterator->intern.data;
1853:         zval_ptr_dtor(obj);
1854:     }
1855:
1856:     /* Otherwise we were called from the owning object free storage handler as
1857:      * it sets iterator->intern.data to IS_UNDEF.
1858:      * We don't even need to destroy iterator->current as we didn't add a
1859:      * reference to it in move_forward or get_iterator */
1860: }
1861: /* }}} */
1862:
1863: /* {{{ spl_filesystem_dir_it_rewind */
1864: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1865: {
1866:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1867:
1868:     IF (!IS_ISNULL(iterator->intern.data)) {
1869:         zval *obj = iterator->intern.data;
1870:         zval_ptr_dtor(obj);
1871:     }
1872:
1873:     /* Otherwise we were called from the owning object free storage handler as
1874:      * it sets iterator->intern.data to IS_UNDEF.
1875:      * We don't even need to destroy iterator->current as we didn't add a
1876:      * reference to it in move_forward or get_iterator */
1877: }
1878: /* }}} */
1879:
1880: /* {{{ spl_filesystem_dir_it_rewind */
1881: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1882: {
1883:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1884:
1885:     IF (!IS_ISNULL(iterator->intern.data)) {
1886:         zval *obj = iterator->intern.data;
1887:         zval_ptr_dtor(obj);
1888:     }
1889:
1890:     /* Otherwise we were called from the owning object free storage handler as
1891:      * it sets iterator->intern.data to IS_UNDEF.
1892:      * We don't even need to destroy iterator->current as we didn't add a
1893:      * reference to it in move_forward or get_iterator */
1894: }
1895: /* }}} */
1896:
1897: /* {{{ spl_filesystem_dir_it_rewind */
1898: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1899: {
1900:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1901:
1902:     IF (!IS_ISNULL(iterator->intern.data)) {
1903:         zval *obj = iterator->intern.data;
1904:         zval_ptr_dtor(obj);
1905:     }
1906:
1907:     /* Otherwise we were called from the owning object free storage handler as
1908:      * it sets iterator->intern.data to IS_UNDEF.
1909:      * We don't even need to destroy iterator->current as we didn't add a
1910:      * reference to it in move_forward or get_iterator */
1911: }
1912: /* }}} */
1913:
1914: /* {{{ spl_filesystem_dir_it_rewind */
1915: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1916: {
1917:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1918:
1919:     IF (!IS_ISNULL(iterator->intern.data)) {
1920:         zval *obj = iterator->intern.data;
1921:         zval_ptr_dtor(obj);
1922:     }
1923:
1924:     /* Otherwise we were called from the owning object free storage handler as
1925:      * it sets iterator->intern.data to IS_UNDEF.
1926:      * We don't even need to destroy iterator->current as we didn't add a
1927:      * reference to it in move_forward or get_iterator */
1928: }
1929: /* }}} */
1930:
1931: /* {{{ spl_filesystem_dir_it_rewind */
1932: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1933: {
1934:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1935:
1936:     IF (!IS_ISNULL(iterator->intern.data)) {
1937:         zval *obj = iterator->intern.data;
1938:         zval_ptr_dtor(obj);
1939:     }
1940:
1941:     /* Otherwise we were called from the owning object free storage handler as
1942:      * it sets iterator->intern.data to IS_UNDEF.
1943:      * We don't even need to destroy iterator->current as we didn't add a
1944:      * reference to it in move_forward or get_iterator */
1945: }
1946: /* }}} */
1947:
1948: /* {{{ spl_filesystem_dir_it_rewind */
1949: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1950: {
1951:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1952:
1953:     IF (!IS_ISNULL(iterator->intern.data)) {
1954:         zval *obj = iterator->intern.data;
1955:         zval_ptr_dtor(obj);
1956:     }
1957:
1958:     /* Otherwise we were called from the owning object free storage handler as
1959:      * it sets iterator->intern.data to IS_UNDEF.
1960:      * We don't even need to destroy iterator->current as we didn't add a
1961:      * reference to it in move_forward or get_iterator */
1962: }
1963: /* }}} */
1964:
1965: /* {{{ spl_filesystem_dir_it_rewind */
1966: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1967: {
1968:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1969:
1970:     IF (!IS_ISNULL(iterator->intern.data)) {
1971:         zval *obj = iterator->intern.data;
1972:         zval_ptr_dtor(obj);
1973:     }
1974:
1975:     /* Otherwise we were called from the owning object free storage handler as
1976:      * it sets iterator->intern.data to IS_UNDEF.
1977:      * We don't even need to destroy iterator->current as we didn't add a
1978:      * reference to it in move_forward or get_iterator */
1979: }
1980: /* }}} */
1981:
1982: /* {{{ spl_filesystem_dir_it_rewind */
1983: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1984: {
1985:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1986:
1987:     IF (!IS_ISNULL(iterator->intern.data)) {
1988:         zval *obj = iterator->intern.data;
1989:         zval_ptr_dtor(obj);
1990:     }
1991:
1992:     /* Otherwise we were called from the owning object free storage handler as
1993:      * it sets iterator->intern.data to IS_UNDEF.
1994:      * We don't even need to destroy iterator->current as we didn't add a
1995:      * reference to it in move_forward or get_iterator */
1996: }
1997: /* }}} */
1998:
1999: /* {{{ spl_filesystem_dir_it_rewind */
2000: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2001: {
2002:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2003:
2004:     IF (!IS_ISNULL(iterator->intern.data)) {
2005:         zval *obj = iterator->intern.data;
2006:         zval_ptr_dtor(obj);
2007:     }
2008:
2009:     /* Otherwise we were called from the owning object free storage handler as
2010:      * it sets iterator->intern.data to IS_UNDEF.
2011:      * We don't even need to destroy iterator->current as we didn't add a
2012:      * reference to it in move_forward or get_iterator */
2013: }
2014: /* }}} */
2015:
2016: /* {{{ spl_filesystem_dir_it_rewind */
2017: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2018: {
2019:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2020:
2021:     IF (!IS_ISNULL(iterator->intern.data)) {
2022:         zval *obj = iterator->intern.data;
2023:         zval_ptr_dtor(obj);
2024:     }
2025:
2026:     /* Otherwise we were called from the owning object free storage handler as
2027:      * it sets iterator->intern.data to IS_UNDEF.
2028:      * We don't even need to destroy iterator->current as we didn't add a
2029:      * reference to it in move_forward or get_iterator */
2030: }
2031: /* }}} */
2032:
2033: /* {{{ spl_filesystem_dir_it_rewind */
2034: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2035: {
2036:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2037:
2038:     IF (!IS_ISNULL(iterator->intern.data)) {
2039:         zval *obj = iterator->intern.data;
2040:         zval_ptr_dtor(obj);
2041:     }
2042:
2043:     /* Otherwise we were called from the owning object free storage handler as
2044:      * it sets iterator->intern.data to IS_UNDEF.
2045:      * We don't even need to destroy iterator->current as we didn't add a
2046:      * reference to it in move_forward or get_iterator */
2047: }
2048: /* }}} */
2049:
2050: /* {{{ spl_filesystem_dir_it_rewind */
2051: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2052: {
2053:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2054:
2055:     IF (!IS_ISNULL(iterator->intern.data)) {
2056:         zval *obj = iterator->intern.data;
2057:         zval_ptr_dtor(obj);
2058:     }
2059:
2060:     /* Otherwise we were called from the owning object free storage handler as
2061:      * it sets iterator->intern.data to IS_UNDEF.
2062:      * We don't even need to destroy iterator->current as we didn't add a
2063:      * reference to it in move_forward or get_iterator */
2064: }
2065: /* }}} */
2066:
2067: /* {{{ spl_filesystem_dir_it_rewind */
2068: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2069: {
2070:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2071:
2072:     IF (!IS_ISNULL(iterator->intern.data)) {
2073:         zval *obj = iterator->intern.data;
2074:         zval_ptr_dtor(obj);
2075:     }
2076:
2077:     /* Otherwise we were called from the owning object free storage handler as
2078:      * it sets iterator->intern.data to IS_UNDEF.
2079:      * We don't even need to destroy iterator->current as we didn't add a
2080:      * reference to it in move_forward or get_iterator */
2081: }
2082: /* }}} */
2083:
2084: /* {{{ spl_filesystem_dir_it_rewind */
2085: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2086: {
2087:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2088:
2089:     IF (!IS_ISNULL(iterator->intern.data)) {
2090:         zval *obj = iterator->intern.data;
2091:         zval_ptr_dtor(obj);
2092:     }
2093:
2094:     /* Otherwise we were called from the owning object free storage handler as
2095:      * it sets iterator->intern.data to IS_UNDEF.
2096:      * We don't even need to destroy iterator->current as we didn't add a
2097:      * reference to it in move_forward or get_iterator */
2098: }
2099: /* }}} */
2100:
2101: /* {{{ spl_filesystem_dir_it_rewind */
2102: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2103: {
2104:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2105:
2106:     IF (!IS_ISNULL(iterator->intern.data)) {
2107:         zval *obj = iterator->intern.data;
2108:         zval_ptr_dtor(obj);
2109:     }
2110:
2111:     /* Otherwise we were called from the owning object free storage handler as
2112:      * it sets iterator->intern.data to IS_UNDEF.
2113:      * We don't even need to destroy iterator->current as we didn't add a
2114:      * reference to it in move_forward or get_iterator */
2115: }
2116: /* }}} */
2117:
2118: /* {{{ spl_filesystem_dir_it_rewind */
2119: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2120: {
2121:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2122:
2123:     IF (!IS_ISNULL(iterator->intern.data)) {
2124:         zval *obj = iterator->intern.data;
2125:         zval_ptr_dtor(obj);
2126:     }
2127:
2128:     /* Otherwise we were called from the owning object free storage handler as
2129:      * it sets iterator->intern.data to IS_UNDEF.
2130:      * We don't even need to destroy iterator->current as we didn't add a
2131:      * reference to it in move_forward or get_iterator */
2132: }
2133: /* }}} */
2134:
2135: /* {{{ spl_filesystem_dir_it_rewind */
2136: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2137: {
2138:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2139:
2140:     IF (!IS_ISNULL(iterator->intern.data)) {
2141:         zval *obj = iterator->intern.data;
2142:         zval_ptr_dtor(obj);
2143:     }
2144:
2145:     /* Otherwise we were called from the owning object free storage handler as
2146:      * it sets iterator->intern.data to IS_UNDEF.
2147:      * We don't even need to destroy iterator->current as we didn't add a
2148:      * reference to it in move_forward or get_iterator */
2149: }
2150: /* }}} */
2151:
2152: /* {{{ spl_filesystem_dir_it_rewind */
2153: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2154: {
2155:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2156:
2157:     IF (!IS_ISNULL(iterator->intern.data)) {
2158:         zval *obj = iterator->intern.data;
2159:         zval_ptr_dtor(obj);
2160:     }
2161:
2162:     /* Otherwise we were called from the owning object free storage handler as
2163:      * it sets iterator->intern.data to IS_UNDEF.
2164:      * We don't even need to destroy iterator->current as we didn't add a
2165:      * reference to it in move_forward or get_iterator */
2166: }
2167: /* }}} */
2168:
2169: /* {{{ spl_filesystem_dir_it_rewind */
2170: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2171: {
2172:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2173:
2174:     IF (!IS_ISNULL(iterator->intern.data)) {
2175:         zval *obj = iterator->intern.data;
2176:         zval_ptr_dtor(obj);
2177:     }
2178:
2179:     /* Otherwise we were called from the owning object free storage handler as
2180:      * it sets iterator->intern.data to IS_UNDEF.
2181:      * We don't even need to destroy iterator->current as we didn't add a
2182:      * reference to it in move_forward or get_iterator */
2183: }
2184: /* }}} */
2185:
2186: /* {{{ spl_filesystem_dir_it_rewind */
2187: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2188: {
2189:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2190:
2191:     IF (!IS_ISNULL(iterator->intern.data)) {
2192:         zval *obj = iterator->intern.data;
2193:         zval_ptr_dtor(obj);
2194:     }
2195:
2196:     /* Otherwise we were called from the owning object free storage handler as
2197:      * it sets iterator->intern.data to IS_UNDEF.
2198:      * We don't even need to destroy iterator->current as we didn't add a
2199:      * reference to it in move_forward or get_iterator */
2200: }
2201: /* }}} */
2202:
2203: /* {{{ spl_filesystem_dir_it_rewind */
2204: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2205: {
2206:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2207:
2208:     IF (!IS_ISNULL(iterator->intern.data)) {
2209:         zval *obj = iterator->intern.data;
2210:         zval_ptr_dtor(obj);
2211:     }
2212:
2213:     /* Otherwise we were called from the owning object free storage handler as
2214:      * it sets iterator->intern.data to IS_UNDEF.
2215:      * We don't even need to destroy iterator->current as we didn't add a
2216:      * reference to it in move_forward or get_iterator */
2217: }
2218: /* }}} */
2219:
2220: /* {{{ spl_filesystem_dir_it_rewind */
2221: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2222: {
2223:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2224:
2225:     IF (!IS_ISNULL(iterator->intern.data)) {
2226:         zval *obj = iterator->intern.data;
2227:         zval_ptr_dtor(obj);
2228:     }
2229:
2230:     /* Otherwise we were called from the owning object free storage handler as
2231:      * it sets iterator->intern.data to IS_UNDEF.
2232:      * We don't even need to destroy iterator->current as we didn't add a
2233:      * reference to it in move_forward or get_iterator */
2234: }
2235: /* }}} */
2236:
2237: /* {{{ spl_filesystem_dir_it_rewind */
2238: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2239: {
2240:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2241:
2242:     IF (!IS_ISNULL(iterator->intern.data)) {
2243:         zval *obj = iterator->intern.data;
2244:         zval_ptr_dtor(obj);
2245:     }
2246:
2247:     /* Otherwise we were called from the owning object free storage handler as
2248:      * it sets iterator->intern.data to IS_UNDEF.
2249:      * We don't even need to destroy iterator->current as we didn't add a
2250:      * reference to it in move_forward or get_iterator */
2251: }
2252: /* }}} */
2253:
2254: /* {{{ spl_filesystem_dir_it_rewind */
2255: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2256: {
2257:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2258:
2259:     IF (!IS_ISNULL(iterator->intern.data)) {
2260:         zval *obj = iterator->intern.data;
2261:         zval_ptr_dtor(obj);
2262:     }
2263:
2264:     /* Otherwise we were called from the owning object free storage handler as
2265:      * it sets iterator->intern.data to IS_UNDEF.
2266:      * We don't even need to destroy iterator->current as we didn't add a
2267:      * reference to it in move_forward or get_iterator */
2268: }
2269: /* }}} */
2270:
2271: /* {{{ spl_filesystem_dir_it_rewind */
2272: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2273: {
2274:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2275:
2276:     IF (!IS_ISNULL(iterator->intern.data)) {
2277:         zval *obj = iterator->intern.data;
2278:         zval_ptr_dtor(obj);
2279:     }
2280:
2281:     /* Otherwise we were called from the owning object free storage handler as
2282:      * it sets iterator->intern.data to IS_UNDEF.
2283:      * We don't even need to destroy iterator->current as we didn't add a
2284:      * reference to it in move_forward or get_iterator */
2285: }
2286: /* }}} */
2287:
2288: /* {{{ spl_filesystem_dir_it_rewind */
2289: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2290: {
2291:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2292:
2293:     IF (!IS_ISNULL(iterator->intern.data)) {
2294:         zval *obj = iterator->intern.data;
2295:         zval_ptr_dtor(obj);
2296:     }
2297:
2298:     /* Otherwise we were called from the owning object free storage handler as
2299:      * it sets iterator->intern.data to IS_UNDEF.
2300:      * We don't even need to destroy iterator->current as we didn't add a
2301:      * reference to it in move_forward or get_iterator */
2302: }
2303: /* }}} */
2304:
2305: /* {{{ spl_filesystem_dir_it_rewind */
2306: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2307: {
2308:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2309:
2310:     IF (!IS_ISNULL(iterator->intern.data)) {
2311:         zval *obj = iterator->intern.data;
2312:         zval_ptr_dtor(obj);
2313:     }
2314:
2315:     /* Otherwise we were called from the owning object free storage handler as
2316:      * it sets iterator->intern.data to IS_UNDEF.
2317:      * We don't even need to destroy iterator->current as we didn't add a
2318:      * reference to it in move_forward or get_iterator */
2319: }
2320: /* }}} */
2321:
2322: /* {{{ spl_filesystem_dir_it_rewind */
2323: static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
2324: {
2325:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
2326:
2327:     IF (!IS_ISNULL(iterator->intern.data)) {
2328:         zval *obj = iterator->intern.data;
2329:         zval_ptr_dtor(obj);
2330:     }
2331:
2332:     /* Otherwise we were called from the owning object free storage handler as
2333:      * it sets iterator->intern.data to IS_UNDEF.
2334:      * We don't even need to destroy iterator->current as we didn't add a
2335:      * reference to it in move_forward or get_iterator */
2336: }
2337: /* }}} */
2338:
2339: /* {{{ spl_filesystem_dir_it_rewind */
23
```

```

1878: ZVAL_NULL(&retval);
1879: return FAILURE;
1880: }
1881: /* }}} */
1882:
1883: /* {{{ declare method parameters */
1884: /* supply a name and default to call by parameter */
1885: ZEND_BEGIN_ARG_INFO(arginfo_info_construct, 0)
1886:     ZEND_ARG_INFO(0, file_name)
1887: ZEND_END_ARG_INFO()
1888:
1889: ZEND_BEGIN_ARG_INFO_KK(arginfo_info_openFile, 0, 0, 0)
1890:     ZEND_ARG_INFO(0, open_mode)
1891:     ZEND_ARG_INFO(0, use_include_path)
1892:     ZEND_ARG_INFO(0, context)
1893: ZEND_END_ARG_INFO()
1894:
1895: ZEND_BEGIN_ARG_INFO_KK(arginfo_info_optimalFileInfoClass, 0, 0, 0)
1896:     ZEND_ARG_INFO(0, class_name)
1897: ZEND_END_ARG_INFO()
1898:
1899: ZEND_BEGIN_ARG_INFO_KK(arginfo_info_optimalSuffix, 0, 0, 0)
1900:     ZEND_ARG_INFO(0, suffix)
1901: ZEND_END_ARG_INFO()
1902:
1903: ZEND_BEGIN_ARG_INFO(arginfo_splFileInfo_void, 0)
1904: ZEND_END_ARG_INFO()
1905:
1906: /* the method table */
1907: /* each method can have its own parameters and visibility */
1908: static const zend_function_entry splFileInfo_functions[] = {
1909:     SPL_ME(splFileInfo, __construct, arginfo_info_construct, ZEND_ACC_PUBLIC)
1910:     SPL_ME(splFileInfo, getPath, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1911:     SPL_ME(splFileInfo, getFilename, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1912:     SPL_ME(splFileInfo, getExtension, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1913:     SPL_ME(splFileInfo, getBasename, arginfo_optimalSuffix, ZEND_ACC_PUBLIC)
1914:     SPL_ME(splFileInfo, getPathname, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1915:     SPL_ME(splFileInfo, getPerms, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1916:     SPL_ME(splFileInfo, getInode, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1917:     SPL_ME(splFileInfo, getSize, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1918:     SPL_ME(splFileInfo, getOwner, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1919:     SPL_ME(splFileInfo, getGroup, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1920:     SPL_ME(splFileInfo, getMTime, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1921:     SPL_ME(splFileInfo, getCTime, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1922:     SPL_ME(splFileInfo, getTyp, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1923:     SPL_ME(splFileInfo, isWritable, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1924:     SPL_ME(splFileInfo, isExecutable, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1925:     SPL_ME(splFileInfo, isReadable, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1926:     SPL_ME(splFileInfo, isExecutable, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1927:     SPL_ME(splFileInfo, isFile, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1928:     SPL_ME(splFileInfo, isDir, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1929:     SPL_ME(splFileInfo, isLink, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1930:     SPL_ME(splFileInfo, getLinkTarget, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1931:     IF HAVE_REALPATH 1 defined(OT)
1932:     SPL_ME(splFileInfo, getRealPath, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1933: #endif
1934:     SPL_ME(splFileInfo, getFileInfo, arginfo_info_optimalFileInfoClass, ZEND_ACC_PUBLIC)
1935:     SPL_ME(splFileInfo, getPathInfo, arginfo_info_optimalFileInfoClass, ZEND_ACC_PUBLIC)
1936:     SPL_ME(splFileInfo, openFile, arginfo_info_openFile, ZEND_ACC_PUBLIC)
1937:     SPL_ME(splFileInfo, setFileInfoClass, arginfo_info_optimalFileInfoClass, ZEND_ACC_PUBLIC)
1938:     SPL_ME(splFileInfo, setFileInfoClass, arginfo_info_optimalFileInfoClass, ZEND_ACC_PUBLIC)
1939:     SPL_ME(splFileInfo, __setState_MW_NULL, __setState_MW_NULL, ZEND_ACC_FINAL)
1940:     SPL_ME(splFileInfo, __toString, splFileInfo, getFilename, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1941:     PHP_FE_END
1942: };
1943:
1944: ZEND_BEGIN_ARG_INFO(arginfo_dir_construct, 0)
1945:     ZEND_ARG_INFO(0, path)
1946: ZEND_END_ARG_INFO()
1947:
1948: ZEND_BEGIN_ARG_INFO(arginfo_dir_iter_seek, 0)
1949:     ZEND_ARG_INFO(0, position)
1950: ZEND_END_ARG_INFO()
1951:
1952: /* the method table */
1953: /* each method can have its own parameters and visibility */
1954: static const zend_function_entry splDirectoryIterator_functions[] = {
1955:     SPL_ME(DirectoryIterator, __construct, arginfo_dir_construct, ZEND_ACC_PUBLIC)
1956:     SPL_ME(DirectoryIterator, getFilename, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1957:     SPL_ME(DirectoryIterator, getExtension, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1958:     SPL_ME(DirectoryIterator, getBasename, arginfo_optimalSuffix, ZEND_ACC_PUBLIC)
1959:     SPL_ME(DirectoryIterator, isValid, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1960:     SPL_ME(DirectoryIterator, rewind, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1961:     SPL_ME(DirectoryIterator, valid, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1962:     SPL_ME(DirectoryIterator, key, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1963:     SPL_ME(DirectoryIterator, current, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1964:     SPL_ME(DirectoryIterator, next, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1965:     SPL_ME(DirectoryIterator, next, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1966:     SPL_ME(DirectoryIterator, __toString, DirectoryIterator, getFilename, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1967:     PHP_FE_END
1968: };
1969:
1970: ZEND_BEGIN_ARG_INFO_KK(arginfo_dir_construct, 0, 0, 1)
1971:     ZEND_ARG_INFO(0, path)
1972:     ZEND_ARG_INFO(0, flags)
1973: ZEND_END_ARG_INFO()
1974:
1975: ZEND_BEGIN_ARG_INFO_KK(arginfo_dir_hasChildren, 0, 0, 0)
1976:     ZEND_ARG_INFO(0, allow_links)
1977: ZEND_END_ARG_INFO()
1978:
1979: ZEND_BEGIN_ARG_INFO_KK(arginfo_dir_setFlags, 0, 0, 0)
1980:     ZEND_ARG_INFO(0, flags)
1981: ZEND_END_ARG_INFO()
1982:
1983: static const zend_function_entry splFilesystemIterator_functions[] = {
1984:     SPL_ME(FilesystemIterator, __construct, arginfo_dir_construct, ZEND_ACC_PUBLIC)
1985:     SPL_ME(FilesystemIterator, rewind, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1986:     SPL_ME(FilesystemIterator, next, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1987:     SPL_ME(FilesystemIterator, key, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1988:     SPL_ME(FilesystemIterator, current, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1989:     SPL_ME(FilesystemIterator, getFlags, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1990:     SPL_ME(FilesystemIterator, setFlags, arginfo_dir_setFlags, ZEND_ACC_PUBLIC)
1991:     PHP_FE_END
1992: };
1993:
1994: static const zend_function_entry splRecursiveDirectoryIterator_functions[] = {
1995:     SPL_ME(RecursiveDirectoryIterator, __construct, arginfo_dir_construct, ZEND_ACC_PUBLIC)
1996:     SPL_ME(RecursiveDirectoryIterator, hasChildren, arginfo_dir_hasChildren, ZEND_ACC_PUBLIC)
1997:     SPL_ME(RecursiveDirectoryIterator, getChildren, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1998:     SPL_ME(RecursiveDirectoryIterator, getSubPath, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1999:     SPL_ME(RecursiveDirectoryIterator, getSubPathname, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
2000:     PHP_FE_END
2001: };
2002:
2003: #if HAVE_GLOB
2004: static const zend_function_entry splGlobIterator_functions[] = {
2005:     SPL_ME(GlobIterator, __construct, arginfo_dir_construct, ZEND_ACC_PUBLIC)
2006:     SPL_ME(GlobIterator, count, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
2007:     PHP_FE_END
2008: };
2009: #endif
2010:
2011:
2012: static int spl_filesystem_file_read(spl_filesystem_object *intern, int silent) /* {{{ */
2013: {
2014:     char *buf;
2015:     size_t line_len = 0;
2016:     zend_long line_add = (intern->u.file.current_line || IS_UNDEF(intern->u.file.current_val)) ? 1 : 0;
2017:
2018:     spl_filesystem_file_free_line(intern);
2019:
2020:     if (php_stream_eof(intern->u.file.stream)) {
2021:         if (!silent) {
2022:             zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot read from file %s", intern->file_name);
2023:         }
2024:         return FAILURE;
2025:     }
2026:
2027:     if (intern->u.file.max_line_len > 0) {
2028:         buf = safe_emalloc(intern->u.file.max_line_len + 1, sizeof(char), 0);
2029:         if (php_stream_get_line(intern->u.file.stream, buf, intern->u.file.max_line_len + 1, sizeof(char)) == NULL) {
2030:             efree(buf);
2031:             buf = NULL;
2032:         } else {
2033:             buf[line_len] = '\0';
2034:         }
2035:     } else {
2036:         buf = php_stream_get_line(intern->u.file.stream, NULL, 0, sizeof(char));
2037:     }
2038:
2039:     if (!buf) {
2040:         intern->u.file.current_line = estrdup("");
2041:         intern->u.file.current_line_len = 0;
2042:     } else {
2043:         if (SPL_BAS_FLAG(intern->flags, SPL_FILE_OBJECT_DROP_NEW_LINE)) {
2044:             line_len = strlen(buf, "\n");
2045:             buf[line_len] = '\0';
2046:         }
2047:
2048:         intern->u.file.current_line = buf;
2049:         intern->u.file.current_line_len = line_len;
2050:     }
2051:     intern->u.file.current_line_num += line_add;
2052:
2053:     return SUCCESS;
2054: } /* }}} */
2055:
2056: static int spl_filesystem_file_call(spl_filesystem_object *intern, zend_function *func_ptr, int pass_num_args, zval *return_value, zval *args) /* {{{ */
2057: {
2058:     zend_fcall_info fci;
2059:     zend_fcall_info_cache fci_cache;
2060:     zval *resource_ptr = intern->u.file.resource, &retval;
2061:     int result;
2062:     int num_args = pass_num_args + (args ? 2 : 1);
2063:     zval *params = (zval *)safe_emalloc(num_args, sizeof(zval), 0);
2064:
2065:     params[0] = *resource_ptr;
2066:
2067:     if (args) {
2068:         if (args[1] == *args;
2069:         )
2070:         {
2071:             if (zend_get_parameters_array_ex(pass_num_args, params + (args ? 2 : 1)) != SUCCESS) {
2072:                 efree(params);
2073:                 return FAILURE;
2074:             }
2075:
2076:             ZVAL_UNDEF(&retval);
2077:
2078:             fci.size = sizeof(fci);
2079:             fci.objekt = NULL;
2080:             fci.retval = &retval;
2081:             fci.param_count = num_args;
2082:             fci.params = params;
2083:             fci.separation = 1;
2084:             ZVAL_STR(&fci.function_name, func_ptr->common.function_name);
2085:
2086:             fci.function_handler = func_ptr;
2087:             fci.calling_scope = NULL;
2088:             fci.called_scope = NULL;
2089:             fci.object = NULL;
2090:
2091:             result = zend_call_function(&fci, &fci_cache);
2092:
2093:             if (result == FAILURE || IS_UNDEF(&retval)) {
2094:                 RETURN_FALSE;
2095:             } else {
2096:                 ZVAL_ZVAL(&return_value, &retval, 0, 0);
2097:             }
2098:
2099:             efree(params);
2100:             return result;
2101:         } /* }}} */
2102:
2103:         #define FileFunctionCall(func_name, pass_num_args, args) /* {{{ */
2104:         {
2105:             zend_function *func_ptr;
2106:             func_ptr = zend_hash_str_find_ptr(CG(function_table), &func_name, sizeof(&func_name) - 1);
2107:             if (func_ptr == NULL) {
2108:                 zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Internal error, function '%s' not found. Please report", &func_name);
2109:                 return;
2110:             }
2111:             spl_filesystem_file_call(intern, func_ptr, pass_num_args, return_value, args);
2112:         } /* }}} */
2113:
2114:         static int spl_filesystem_file_read_csv(spl_filesystem_object *intern, char delimiter, char enclosure, char escape, zval *return_value) /* {{{ */
2115:         {
2116:             int ret = SUCCESS;
2117:             zval *value;
2118:
2119:             do {
2120:                 spl_filesystem_file_read(intern, 1);
2121:                 while (ret == SUCCESS & intern->u.file.current_line_len & SPL_BAS_FLAG(intern->flags, SPL_FILE_OBJECT_SKIP_EMPTY));
2122:                 if (ret == SUCCESS) {
2123:                     size_t buf_len = intern->u.file.current_line_len;
2124:                     char *buf = estrdup(intern->u.file.current_line, buf_len);
2125:
2126:                     if (!IS_UNDEF(intern->u.file.current_val)) {
2127:                         zval_ptr_stor(intern->u.file.current_val);
2128:                         ZVAL_UNDEF(intern->u.file.current_val);
2129:                     }
2130:
2131:                     php_getcsv(intern->u.file.stream, delimiter, enclosure, escape, buf_len, buf, intern->u.file.current_val);
2132:                     spl_ptr_stor(return_value);
2133:                     value = &intern->u.file.current_val;
2134:                     ZVAL_COPY(&return_value, value);
2135:                 }
2136:             }
2137:             return ret;
2138:         }
2139:         /* }}} */
2140:
2141:         static int spl_filesystem_file_read_line_ex(spl_ptr, spl_filesystem_object *intern, int silent) /* {{{ */
2142:         {
2143:             /* 1) use fgets(2) overlaid call the function, 3) do it directly */
2144:             if (SPL_BAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_CSV) || intern->u.file.func_ptr == &common.spl_ptr_stor(spl_ptr_stor)) {
2145:                 if (php_stream_eof(intern->u.file.stream)) {
2146:                     if (!silent) {
2147:                         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot read from file %s", intern->file_name);
2148:                     }
2149:                     return FAILURE;
2150:                 }
2151:                 if (SPL_BAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_CSV)) {
2152:                     return spl_filesystem_file_read_csv(intern, intern->u.file.delimiter, intern->u.file.enclosure, intern->u.file.escape, NULL);
2153:                 } else {
2154:                     zend_execute_data *execute_data = EG(current_execute_data);
2155:                     zend_call_method_with_0_params(this_ptr, &OBJCE_X(THIS), intern->u.file.func_ptr, "getCurrentLine", &retval);
2156:                     if (!IS_UNDEF(&retval)) {
2157:                         if (intern->u.file.current_line || IS_UNDEF(intern->u.file.current_val)) {
2158:                             intern->u.file.current_line_num++;
2159:                         }
2160:                         spl_filesystem_file_free_line(intern);
2161:                         if (!IS_STRING(&retval)) {
2162:                             intern->u.file.current_line = estrdup(IS_STRING(&retval), 2 * STRLEN(&retval));
2163:                             if (!IS_UNDEF(intern->u.file.current_val)) {
2164:                                 zval value = &retval;
2165:                                 ZVAL_COPY(&intern->u.file.current_val, value);
2166:                             }
2167:                             spl_ptr_stor(&retval);
2168:                             return SUCCESS;
2169:                         } else {
2170:                             return FAILURE;
2171:                         }
2172:                     } else {
2173:                         if (IS_UNDEF(intern->u.file.current_val)) {
2174:                             return FAILURE;
2175:                         }
2176:                     }
2177:                 }
2178:             }
2179:             return FAILURE;
2180:         }
2181:         /* }}} */
2182:
2183:         static int spl_filesystem_file_is_empty_line(spl_ptr, spl_filesystem_object *intern) /* {{{ */
2184:         {
2185:             if (intern->u.file.current_line) {
2186:                 return intern->u.file.current_line_len == 0;
2187:             } else if (IS_UNDEF(intern->u.file.current_val)) {
2188:                 if (IS_STRING(&intern->u.file.current_val)) {
2189:                     return IS_STRING(&intern->u.file.current_val) == 0;
2190:                 } else {
2191:                     return IS_STRING(&intern->u.file.current_val) == 0;
2192:                 }
2193:             } else {
2194:                 return IS_STRING(&intern->u.file.current_val) == 0;
2195:             }
2196:             if (SPL_BAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_CSV) && zend_hash_num_elements(&INTERNAL(intern->u.file.current_val)) == 1) {
2197:                 if (IS_STRING(&intern->u.file.current_val)) {
2198:                     return IS_STRING(&intern->u.file.current_val) == 0;
2199:                 }
2200:             }
2201:             while (IS_UNDEF(&INTERNAL(intern->u.file.current_val)->data[0]) || IS_UNDEF(&INTERNAL(intern->u.file.current_val)->data[1])) {
2202:                 if (IS_UNDEF(&INTERNAL(intern->u.file.current_val)->data[0]) || IS_UNDEF(&INTERNAL(intern->u.file.current_val)->data[1])) {
2203:                     return IS_STRING(&INTERNAL(intern->u.file.current_val)->data[0]) == 0;
2204:                 }
2205:             }
2206:             return IS_STRING(&INTERNAL(intern->u.file.current_val)->data[0]) == 0;
2207:         }
2208:         /* }}} */
2209:
2210:         static int spl_filesystem_file_read_line(spl_ptr, spl_filesystem_object *intern, int silent) /* {{{ */
2211:         {
2212:             int ret = spl_filesystem_file_read_line_ex(this_ptr, intern, silent);
2213:             while (SPL_BAS_FLAG(intern->flags, SPL_FILE_OBJECT_SKIP_EMPTY) && ret == SUCCESS & spl_filesystem_file_is_empty_line(intern)) {
2214:                 spl_filesystem_file_free_line(intern);
2215:                 ret = spl_filesystem_file_read_line_ex(this_ptr, intern, silent);
2216:             }
2217:             return ret;
2218:         }
2219:         /* }}} */
2220:
2221:         static void spl_filesystem_file_rewind(spl_ptr, spl_filesystem_object *intern) /* {{{ */
2222:         {
2223:             if (!intern->u.file.stream) {
2224:                 zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2225:                 return;
2226:             }
2227:             if (!spl_ptr_stor_rewind(intern->u.file.stream)) {
2228:                 zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot rewind file %s", intern->file_name);
2229:             }
2230:             spl_filesystem_file_free_line(intern);
2231:             intern->u.file.current_line_num = 0;
2232:             if (SPL_BAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_READ)) {
2233:                 spl_filesystem_file_read_line(this_ptr, intern, 1);
2234:             }
2235:         }
2236:         /* }}} */
2237:
2238:         /* {{{ proto void SplFileObject::__construct(string filename [, string mode = 'r'], bool use_include_path [, resource context]) */
2239:         /* Construct a new file object */
2240:         /* Construct a new file object */
2241:         /* Construct a new file object */
2242:         /* Construct a new file object */
2243:         /* Construct a new file object */
2244:         /* Construct a new file object */
2245:         /* Construct a new file object */
2246:         /* Construct a new file object */
2247:         /* Construct a new file object */
2248:         /* Construct a new file object */
2249:         /* Construct a new file object */
2250:         /* Construct a new file object */
2251:         /* Construct a new file object */
2252:         /* Construct a new file object */
2253:         /* Construct a new file object */
2254:         /* Construct a new file object */
2255:         /* Construct a new file object */
2256:         /* Construct a new file object */
2257:         /* Construct a new file object */
2258:         /* Construct a new file object */
2259:         /* Construct a new file object */
2260:         /* Construct a new file object */
2261:         /* Construct a new file object */
2262:         /* Construct a new file object */
2263:         /* Construct a new file object */
2264:         /* Construct a new file object */
2265:         /* Construct a new file object */
2266:         /* Construct a new file object */
2267:         /* Construct a new file object */
2268:         /* Construct a new file object */
2269:         /* Construct a new file object */
2270:         /* Construct a new file object */
2271:         /* Construct a new file object */
2272:         /* Construct a new file object */
2273:         /* Construct a new file object */
2274:         /* Construct a new file object */
2275:         /* Construct a new file object */
2276:         /* Construct a new file object */
2277:         /* Construct a new file object */
2278:         /* Construct a new file object */
2279:         /* Construct a new file object */
2280:         /* Construct a new file object */
2281:         /* Construct a new file object */
2282:         /* Construct a new file object */
2283:         /* Construct a new file object */
2284:         /* Construct a new file object */
2285:         /* Construct a new file object */
2286:         /* Construct a new file object */
2287:         /* Construct a new file object */
2288:         /* Construct a new file object */
2289:         /* Construct a new file object */
2290:         /* Construct a new file object */
2291:         /* Construct a new file object */
2292:         /* Construct a new file object */
2293:         /* Construct a new file object */
2294:         /* Construct a new file object */
2295:         /* Construct a new file object */
2296:         /* Construct a new file object */
2297:         /* Construct a new file object */
2298:         /* Construct a new file object */
2299:         /* Construct a new file object */
2300:         /* Construct a new file object */
2301:         /* Construct a new file object */
2302:         /* Construct a new file object */
2303:         /* Construct a new file object */
2304:         /* Construct a new file object */
2305:         /* Construct a new file object */
2306:         /* Construct a new file object */
2307:         /* Construct a new file object */
2308:         /* Construct a new file object */
2309:         /* Construct a new file object */
2310:         /* Construct a new file object */
2311:         /* Construct a new file object */
2312:         /* Construct a new file object */
2313:         /* Construct a new file object */
2314:         /* Construct a new file object */
2315:         /* Construct a new file object */
2316:         /* Construct a new file object */
2317:         /* Construct a new file object */
2318:         /* Construct a new file object */
2319:         /* Construct a new file object */
2320:         /* Construct a new file object */
2321:         /* Construct a new file object */
2322:         /* Construct a new file object */
2323:         /* Construct a new file object */
2324:         /* Construct a new file object */
2325:         /* Construct a new file object */
2326:         /* Construct a new file object */
2327:         /* Construct a new file object */
2328:         /* Construct a new file object */
2329:         /* Construct a new file object */
2330:         /* Construct a new file object */
2331:         /* Construct a new file object */
2332:         /* Construct a new file object */
2333:         /* Construct a new file object */
2334:         /* Construct a new file object */
2335:         /* Construct a new file object */
2336:         /* Construct a new file object */
2337:         /* Construct a new file object */
2338:         /* Construct a new file object */
2339:         /* Construct a new file object */
2340:         /* Construct a new file object */
2341:         /* Construct a new file object */
2342:         /* Construct a new file object */
2343:         /* Construct a new file object */
2344:         /* Construct a new file object */
2345:         /* Construct a new file object */
2346:         /* Construct a new file object */
2347:         /* Construct a new file object */
2348:         /* Construct a new file object */
2349:         /* Construct a new file object */
2350:         /* Construct a new file object */
2351:         /* Construct a new file object */
2352:         /* Construct a new file object */
2353:         /* Construct a new file object */
2354:         /* Construct a new file object */
2355:         /* Construct a new file object */
2356:         /* Construct a new file object */
2357:         /* Construct a new file object */
2358:         /* Construct a new file object */
2359:         /* Construct a new file object */
2360:         /* Construct a new file object */
2361:         /* Construct a new file object */
2362:         /* Construct a new file object */
2363:         /* Construct a new file object */
2364:         /* Construct a new file object */
2365:         /* Construct a new file object */
2366:         /* Construct a new file object */
2367:         /* Construct a new file object */
2368:         /* Construct a new file object */
2369:         /* Construct a new file object */
2370:         /* Construct a new file object */
2371:         /* Construct a new file object */
2372:         /* Construct a new file object */
2373:         /* Construct a new file object */
2374:         /* Construct a new file object */
2375:         /* Construct a new file object */
2376:         /* Construct a new file object */
2377:         /* Construct a new file object */
2378:         /* Construct a new file object */
2379:         /* Construct a new file object */
2380:         /* Construct a new file object */
2381:         /* Construct a new file object */
2382:         /* Construct a new file object */
2383:         /* Construct a new file object */
2384:         /* Construct a new file object */
2385:         /* Construct a new file object */
2386:         /* Construct a new file object */
2387:         /* Construct a new file object */
2388:         /* Construct a new file object */
2389:         /* Construct a new file object */
2390:         /* Construct a new file object */
2391:         /* Construct a new file object */
2392:         /* Construct a new file object */
2393:         /* Construct a new file object */
2394:         /* Construct a new file object */
2395:         /* Construct a new file object */
2396:         /* Construct a new file object */
2397:         /* Construct a new file object */
2398:         /* Construct a new file object */
2399:         /* Construct a new file object */
2400:         /* Construct a new file object */
2401:         /* Construct a new file object */
2402:         /* Construct a new file object */
2403:         /* Construct a new file object */
2404:         /* Construct a new file object */
2405:         /* Construct a new file object */
2406:         /* Construct a new file object */
2407:         /* Construct a new file object */
2408:         /* Construct a new file object */
2409:         /* Construct a new file object */
2410:         /* Construct a new file object */
2411:         /* Construct a new file object */
2412:         /* Construct a new file object */
2413:         /* Construct a new file object */
2414:         /* Construct a new file object */
2415:         /* Construct a new file object */
2416:         /* Construct a new file object */
2417:         /* Construct a new file object */
2418:         /* Construct a new file object */
2419:         /* Construct a new file object */
2420:         /* Construct a new file object */
2421:         /* Construct a new file object */
2422:         /* Construct a new file object */
2423:         /* Construct a new file object */
2424:         /* Construct a new file object */
2425:         /* Construct a new file object */
2426:         /* Construct a new file object */
2427:         /* Construct a new file object */
2428:         /* Construct a new file object */
2429:         /* Construct a new file object */
2430:         /* Construct a new file object */
2431:         /* Construct a new file object */
2432:         /* Construct a new file object */
2433:         /* Construct a new file object */
2434:         /* Construct a new file object */
2435:         /* Construct a new file object */
2436:         /* Construct a new file object */
2437:         /* Construct a new file object */
2438:         /* Construct a new file object */
2439:         /* Construct a new file object */
2440:         /* Construct a new file object */
2441:         /* Construct a new file object */
2442:         /* Construct a new file object */
2443:         /* Construct a new file object */
2444:         /* Construct a new file object */
2445:         /* Construct a new file object */
2446:         /* Construct a new file object */
2447:         /* Construct a new file object */
2448:         /* Construct a new file object */
2449:         /* Construct a new file object */
2450:         /* Construct a new file object */
2451:         /* Construct a new file object */
2452:         /* Construct a new file object */
2453:         /* Construct a new file object */
2454:         /* Construct a new file object */
2455:         /* Construct a new file object */
2456:         /* Construct a new file object */
2457:         /* Construct a new file object */
2458:         /* Construct a new file object */
2459:         /* Construct a new file object */
2460:         /* Construct a new file object */
2461:         /* Construct a new file object */
2462:         /* Construct a new file object */
2463:         /* Construct a new file object */
2464:         /* Construct a new file object */
2465:         /* Construct a new file object */
2466:         /* Construct a new file object */
2467:         /* Construct a new file object */
2468:         /* Construct a new file object */
2469:         /* Construct a new file object */
2470:         /* Construct a new file object */
2471:         /* Construct a new file object */
2472:         /* Construct a new file object */
2473:         /* Construct a new file object */
2474:         /* Construct a new file object */
2475:         /* Construct a new file object */
2476:         /* Construct a new file object */
2477:         /* Construct a new file object */
2478:         /* Construct a new file object */
2479:         /* Construct a new file object */
2480:         /* Construct a new file object */
2481:         /* Construct a new file object */
2482:         /* Construct a new file object */
2483:         /* Construct a new file object */
2484:         /* Construct a new file object */
2485:         /* Construct a new file object */
2486:         /* Construct a new file object */
2487:         /* Construct a new file object */
2488:         /* Construct a new file object */
2489:         /* Construct a new file object */
2490:         /* Construct a new file object */
2491:         /* Construct a new file object */
2492:         /* Construct a new file object */
2493:         /* Construct a new file object */
2494:         /* Construct a new file object */
2495:         /* Construct a new file object */
2496:         /* Construct a new file object */
2497:         /* Construct a new file object */
2498:         /* Construct a new file object */
2499:         /* Construct a new file object */
2500:         /* Construct a new file object */
2501:         /* Construct a new file object */
2502:         /* Construct a new file object */
2503:         /* Construct a new file object */
2504:         /* Construct a new file object */
2505:         /* Construct a new file object */
2506:         /* Construct a new file object */
2507:         /* Construct a new file object */
2508:         /* Construct a new file object */
2509:         /* Construct a new file object */
2510:         /* Construct a new file object */
2511:         /* Construct a new file object */
2512:         /* Construct a new file object */
2513:         /* Construct a new file object */
2514:         /* Construct a new file object */
2515:         /* Construct a new file object */
2516:         /* Construct a new file object */
2517:         /* Construct a new file object */
2518:         /* Construct a new file object */
2519:         /* Construct a new file object */
2520:         /* Construct a new file object */
2521:         /* Construct a new file object */
2522:         /* Construct a new file object */
2523:         /* Construct a new file object */
2524:         /* Construct a new file object */
2525:         /* Construct a new file object */
2526:         /* Construct a new file object */
2527:         /* Construct a new file object */
2528:         /* Construct a new file object */
2529:         /* Construct a new file object */
2530:         /* Construct a new file object */
2531:         /* Construct a new file object */
2532:         /* Construct a new file object */
2533:         /* Construct a new file object */
2534:         /* Construct a new file object */
2535:         /* Construct a new file object */
2536:         /* Construct a new file object */
2537:         /* Construct a new file object */
2538:         /* Construct a new file object */
2539:         /* Construct a new file object */
2540:         /* Construct a new file object */
2541:         /* Construct a new file object */
2542:         /* Construct a new file object */
2543:         /* Construct a new file object */
2544:         /* Construct a new file object */
2545:         /* Construct a new file object */
2546:         /* Construct a new file object */
2547:         /* Construct a new file object */
2548:         /* Construct a new file object */
2549:         /* Construct a new file object */
2550:         /* Construct a new file object */
2551:         /* Construct a new file object */
2552:         /* Construct a new file object */
2553:         /* Construct a new file object */
2554:         /* Construct a new file object */
2555:         /* Construct a new file object */
2556:         /* Construct a new file object */
2557:         /* Construct a new file object */
2558:         /* Construct a new file object */
2559:         /* Construct a new file object */
2560:         /* Construct a new file object */
2561:         /* Construct a new file object */
2562:         /* Construct a new file object */
2563:         /* Construct a new file object */
2564:         /* Construct a new file object */
2565:         /* Construct a new file object */
2566:         /* Construct a new file object */
2567:         /* Construct a new file object */
2568:         /* Construct a new file object */
2569:         /* Construct a new file object */
2570:         /* Construct a new file object */
2571:         /* Construct a new file object */
2572:         /* Construct a new file object */
2573:         /* Construct a new file object */
2574:         /* Construct a new file object */
2575:         /* Construct a new file object */
2576:         /* Construct a new file object */
2577:         /* Construct a new file object */
2578:         /* Construct a new file object */
2579:         /* Construct a new file object */
2580:         /* Construct a new file object */
2581:         /* Construct a new file object */
2582:         /* Construct a new file object */
2583:         /* Construct a new file object */
2584:         /* Construct a new file object */
2585:         /* Construct a new file object */
2586:         /* Construct a new file object */
2587:         /* Construct a new file object */
2588:         /* Construct a new file object */
2589:         /* Construct a new file object */
2590:         /* Construct a new file object */
2591:         /* Construct a new file object */
2592:         /* Construct a new file object */
2593:         /* Construct a new file object */
2594:         /* Construct a new file object */
2595:         /* Construct a new file object */
2596:         /* Construct a new file object */
2597:         /* Construct a new file object */
2598:         /* Construct a new file object */
2599:         /* Construct a new file object */
2600:         /* Construct a new file object */
2601:         /* Construct a new file object */
2602:         /* Construct a new file object */
2603:         /* Construct a new file object */
2604:         /* Construct a new file object */
2605:         /* Construct a new file object */
2606:         /* Construct a new file object */
2607:         /* Construct a new file object */
2608:         /* Construct a new file object */
2609:         /* Construct a new file object */
2610:         /* Construct a new file object */
2611:         /* Construct a new file object */
2612:         /* Construct a new file object */
2613:         /* Construct a new file object */
2614:         /* Construct a new file object */
2615:         /* Construct a new file object */
2616:         /* Construct a new file object */
2617:         /* Construct a new file object */
2618:         /* Construct a new file object */
2619:         /* Construct a new file object */
2620:         /* Construct a new file object */
2621:         /* Construct a new file object */
2622:         /* Construct a new file object */
2623:         /* Construct a new file object */
2624:         /* Construct a new file object */
2625:         /* Construct a new file object */
2626:         /* Construct a new file object */
2627:         /* Construct a new file object */
2628:         /* Construct a new file object */
2629:         /* Construct a new file object */
2630:         /* Construct a new file object */
2631:         /* Construct a new file object */
2632:         /* Construct a new file object */
2633:         /* Construct a new file object */
2634:         /* Construct a new file object */
2635:         /* Construct a new file object */
2636:         /* Construct a new file object */
2637:         /* Construct a new file object */
2638:         /* Construct a new file object */
2639:         /* Construct a new file object */
2640:         /* Construct a new file object */
2641:         /* Construct a new file object */
2642:         /* Construct a new file object */
2643:         /* Construct a new file object */
2644:         /* Construct a new file object */
2645:         /* Construct a new file object */
2646:         /* Construct a new file object */
2647:         /* Construct a new file object */
2648:         /* Construct a new file object */
2649:         /* Construct a new file object */
2650:         /* Construct a new file object */
2651:         /* Construct a new file object */
2652:         /* Construct a new file object */
2653:         /* Construct a new file object */
2654:         /* Construct a new file object */
2655:         /* Construct a new file object */
2656:         /* Construct a new file object */
2657:         /* Construct a new file object */
2658:         /* Construct a new file object */
2659:         /* Construct a new file object */
2660:         /* Construct a new file object */
2661:         /* Construct a new file object */
2662:         /* Construct a new file object */
2663:         /* Construct a new file object */

```

```

2253: spl_filesystem_object *intern = _Z_SPLFILESYSTEM_P(getThis());
2254: zend_bool use_include_path = 0;
2255: char *pl, *pz;
2256: char *tmp_path;
2257: size_t tmp_path_len;
2258: zend_error_handling error_handling;
2259:
2260: intern->u.file.open_mode = NULL;
2261: intern->u.file.open_mode_len = 0;
2262:
2263: IF (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "p|abst",
2264:     &intern->file_name, &intern->file_name_len,
2265:     &intern->u.file.open_mode, &intern->u.file.open_mode_len,
2266:     &use_include_path, &intern->u.file.sconnext) == FAILURE) {
2267:     intern->u.file.open_mode = NULL;
2268:     &intern->file_name = NULL;
2269:     return;
2270: }
2271:
2272: IF (intern->u.file.open_mode == NULL) {
2273:     intern->u.file.open_mode = "r";
2274:     intern->u.file.open_mode_len = 1;
2275: }
2276:
2277: zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
2278:
2279: IF (spl_filesystem_file_open(intern, use_include_path, 0) == SUCCESS) {
2280:     tmp_path_len = strlen(intern->u.file.stream->orig_path);
2281:
2282:     IF (tmp_path_len > 1 && IS_SLASH_AT(intern->u.file.stream->orig_path, tmp_path_len-1)) {
2283:         tmp_path_len--;
2284:     }
2285:
2286:     tmp_path = estrndup(intern->u.file.stream->orig_path, tmp_path_len);
2287:
2288:     p1 = strchr(tmp_path, '/');
2289:     IF defined(PHP_WIN32)
2290:         p2 = strchr(tmp_path, '\\');
2291:     while
2292:         p2 < p1 && p2 > 0 {
2293:         p2--;
2294:         IF (p1 || p2) {
2295:             intern->u_path_len = ((p1 ? p2 : p1) - tmp_path);
2296:         } ELSE {
2297:             intern->u_path_len = 0;
2298:         }
2299:         ofree(tmp_path);
2300:         p1 = p2;
2301:     }
2302:     intern->u_path = estrndup(intern->u.file.stream->orig_path, intern->u_path_len);
2303: }
2304:
2305: zend_restore_error_handling(&error_handling);
2306:
2307: /* }}} */
2308:
2309: /* {{{ proto void SplFileObject::__construct([int max_memory])
2310:    Construct a new temp file object */
2311: SPL_METHOD(SplFileObject, __construct)
2312: {
2313:     zend_long max_memory = PHP_STREAM_MAX_LEN;
2314:     char tmp_fname[64];
2315:     spl_filesystem_object *intern = _Z_SPLFILESYSTEM_P(getThis());
2316:     zend_error_handling error_handling;
2317:
2318:     IF (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "|l", &max_memory) == FAILURE) {
2319:         return;
2320:     }
2321:
2322:     IF (max_memory < 0) {
2323:         intern->file_name = "php://memory";
2324:         intern->file_name_len = 12;
2325:     } ELSE IF (ZEND_NUM_ARGS() > 1) {
2326:         intern->file_name_len = sprintf(tmp_fname, sizeof(tmp_fname), "php://temp/memmemory:" ZEND_LONG_FMT, max_memory);
2327:         intern->file_name = tmp_fname;
2328:     } ELSE {
2329:         intern->file_name = "php://temp";
2330:         intern->file_name_len = 10;
2331:     }
2332:
2333:     intern->u.file.open_mode = "wb";
2334:     intern->u.file.open_mode_len = 1;
2335:
2336:     zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
2337:     IF (spl_filesystem_file_open(intern, 0, 0) == SUCCESS) {
2338:         intern->u_path_len = 0;
2339:         intern->u_path = estrndup("", 0);
2340:     }
2341:     zend_restore_error_handling(&error_handling);
2342:     /* }}} */
2343:
2344: /* {{{ proto void SplFileObject::rewind()
2345:    Rewind the file and read the first line */
2346: SPL_METHOD(SplFileObject, rewind)
2347: {
2348:     spl_filesystem_object *intern = _Z_SPLFILESYSTEM_P(getThis());
2349:
2350:     IF (zend_parse_parameters_none() == FAILURE) {
2351:         return;
2352:     }
2353:
2354:     spl_filesystem_file_rewind(getThis(), intern);
2355:     /* }}} */
2356:
2357: /* {{{ proto void SplFileObject::eof()
2358:    Return whether end of file is reached */
2359: SPL_METHOD(SplFileObject, eof)
2360: {
2361:     spl_filesystem_object *intern = _Z_SPLFILESYSTEM_P(getThis());
2362:
2363:     IF (zend_parse_parameters_none() == FAILURE) {
2364:         return;
2365:     }
2366:
2367:     IF(!intern->u.file.stream) {
2368:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2369:         return;
2370:     }
2371:
2372:     RETURN_BOOL(pphp_stream_eof(intern->u.file.stream));
2373:     /* }}} */
2374:
2375: /* {{{ proto void SplFileObject::valid()
2376:    Return true if */
2377: SPL_METHOD(SplFileObject, valid)
2378: {
2379:     spl_filesystem_object *intern = _Z_SPLFILESYSTEM_P(getThis());
2380:
2381:     IF (zend_parse_parameters_none() == FAILURE) {
2382:         return;
2383:     }
2384:
2385:     IF (spl_was_flag(intern->flags, SPL_FILE_OBJECT_READ Ahead)) {
2386:         RETURN_BOOL(intern->u.file.current_line || !IS_HUNDERF(intern->u.file.current_val));
2387:     } ELSE {
2388:         IF(!intern->u.file.stream) {
2389:             RETURN_FALSE;
2390:         }
2391:         RETURN_BOOL(pphp_stream_eof(intern->u.file.stream));
2392:     }
2393:     /* }}} */
2394:
2395: /* {{{ proto string SplFileObject::fgets()
2396:    Return next line from file */
2397: SPL_METHOD(SplFileObject, fgets)
2398: {
2399:     spl_filesystem_object *intern = _Z_SPLFILESYSTEM_P(getThis());
2400:
2401:     IF (zend_parse_parameters_none() == FAILURE) {
2402:         return;
2403:     }
2404:
2405:     IF(!intern->u.file.stream) {
2406:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2407:         return;
2408:     }
2409:
2410:     IF (spl_filesystem_file_read(intern, 0) == FAILURE) {
2411:         RETURN_FALSE;
2412:     }
2413:
2414:     RETURN_STRING(intern->u.file.current_line, intern->u.file.current_line_len);
2415:     /* }}} */
2416:
2417: /* {{{ proto string SplFileObject::current()
2418:    Return current line from file */
2419: SPL_METHOD(SplFileObject, current)
2420: {
2421:     spl_filesystem_object *intern = _Z_SPLFILESYSTEM_P(getThis());
2422:
2423:     IF (zend_parse_parameters_none() == FAILURE) {
2424:         return;
2425:     }
2426:
2427:     IF(!intern->u.file.stream) {
2428:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2429:         return;
2430:     }
2431:
2432:     IF (intern->u.file.current_line && !IS_HUNDERF(intern->u.file.current_val)) {
2433:         spl_filesystem_file_read_line(intern, 1);
2434:     }
2435:
2436:     IF (intern->u.file.current_line && !IS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_CSV) || !IS_HUNDERF(intern->u.file.current_val)) {
2437:         RETURN_STRING(intern->u.file.current_line, intern->u.file.current_line_len);
2438:     } ELSE IF (!IS_HUNDERF(intern->u.file.current_val)) {
2439:         eval *value = &intern->u.file.current_val;
2440:         RETURN_ZVAL(value);
2441:     } ELSE IF (IS_HUNDERF(intern->u.file.current_val)) {
2442:         RETURN_ZVAL(return_value, value);
2443:     }
2444: }
2445:
2446:
2447:
2448:
2449:
2450:
2451:
2452:
2453:
2454:
2455:
2456:
2457:
2458:
2459:
2460:
2461:
2462:
2463:
2464:
2465:
2466:
2467:
2468:
2469:
2470:
2471:
2472:
2473:
2474:
2475:
2476:
2477:
2478:
2479:
2480:
2481:
2482:
2483:
2484:
2485:
2486:
2487:
2488:
2489:
2490:
2491:
2492:
2493:
2494:
2495:
2496:
2497:
2498:
2499:
2500:
2501:
2502:
2503:
2504:
2505:
2506:
2507:
2508:
2509:
2510:
2511:
2512:
2513:
2514:
2515:
2516:
2517:
2518:
2519:
2520:
2521:
2522:
2523:
2524:
2525:
2526:
2527:
2528:
2529:
2530:
2531:
2532:
2533:
2534:
2535:
2536:
2537:
2538:
2539:
2540:
2541:
2542:
2543:
2544:
2545:
2546:
2547:
2548:
2549:
2550:
2551:
2552:
2553:
2554:
2555:
2556:
2557:
2558:
2559:
2560:
2561:
2562:
2563:
2564:
2565:
2566:
2567:
2568:
2569:
2570:
2571:
2572:
2573:
2574:
2575:
2576:
2577:
2578:
2579:
2580:
2581:
2582:
2583:
2584:
2585:
2586:
2587:
2588:
2589:
2590:
2591:
2592:
2593:
2594:
2595:
2596:
2597:
2598:
2599:
2600:
2601:
2602:
2603:
2604:
2605:
2606:
2607:
2608:
2609:
2610:
2611:
2612:
2613:
2614:
2615:
2616:
2617:
2618:
2619:
2620:
2621:
2622:
2623:
2624:
2625:
2626:
2627:
2628:
2629:
2630:
2631:
2632:
2633:
2634:
2635:
2636:
2637:
2638:
2639:
2640:
2641:
2642:
2643:
2644:
2645:
2646:
2647:
2648:
2649:
2650:
2651:
2652:
2653:
2654:
2655:
2656:
2657:
2658:
2659:
2660:
2661:
2662:
2663:
2664:
2665:
2666:
2667:
2668:
2669:
2670:
2671:
2672:
2673:
2674:
2675:
2676:
2677:
2678:
2679:
2680:
2681:
2682:
2683:
2684:
2685:
2686:
2687:
2688:
2689:
2690:
2691:
2692:
2693:
2694:
2695:
2696:
2697:
2698:
2699:
2700:
2701:
2702:
2703:
2704:
2705:
2706:
2707:
2708:
2709:
2710:
2711:
2712:
2713:
2714:
2715:
2716:
2717:
2718:
2719:
2720:
2721:
2722:
2723:
2724:
2725:
2726:
2727:
2728:
2729:
2730:
2731:
2732:
2733:
2734:
2735:
2736:
2737:
2738:
2739:
2740:
2741:
2742:
2743:
2744:
2745:
2746:
2747:
2748:
2749:
2750:
2751:
2752:
2753:
2754:
2755:
2756:
2757:
2758:
2759:
2760:
2761:
2762:
2763:
2764:
2765:
2766:
2767:
2768:
2769:
2770:
2771:
2772:
2773:
2774:
2775:
2776:
2777:
2778:
2779:
2780:
2781:
2782:
2783:
2784:
2785:
2786:
2787:
2788:
2789:
2790:
2791:
2792:
2793:
2794:
2795:
2796:
2797:
2798:
2799:
2800:
2801:
2802:
2803:
2804:
2805:
2806:
2807:
2808:
2809:
2810:
2811:
2812:
2813:
2814:
2815:
2816:
2817:
2818:
2819:
2820:
2821:
2822:
2823:
2824:
2825:
2826:
2827:
2828:
2829:
2830:
2831:
2832:
2833:
2834:
2835:
2836:
2837:
2838:
2839:
2840:
2841:
2842:
2843:
2844:
2845:
2846:
2847:
2848:
2849:
2850:
2851:
2852:
2853:
2854:
2855:
2856:
2857:
2858:
2859:
2860:
2861:
2862:
2863:
2864:
2865:
2866:
2867:
2868:
2869:
2870:
2871:
2872:
2873:
2874:
2875:
2876:
2877:
2878:
2879:
2880:
2881:
2882:
2883:
2884:
2885:
2886:
2887:
2888:
2889:
2890:
2891:
2892:
2893:
2894:
2895:
2896:
2897:
2898:
2899:
2900:
2901:
2902:
2903:
2904:
2905:
2906:
2907:
2908:
2909:
2910:
2911:
2912:
2913:
2914:
2915:
2916:
2917:
2918:
2919:
2920:
2921:
2922:
2923:
2924:
2925:
2926:
2927:
2928:
2929:
2930:
2931:
2932:
2933:
2934:
2935:
2936:
2937:
2938:
2939:
2940:
2941:
2942:
2943:
2944:
2945:
2946:
2947:
2948:
2949:
2950:
2951:
2952:
2953:
2954:
2955:
2956:
2957:
2958:
2959:
2960:
2961:
2962:
2963:
2964:
2965:
2966:
2967:
2968:
2969:
2970:
2971:
2972:
2973:
2974:
2975:
2976:
2977:
2978:
2979:
2980:
2981:
2982:
2983:
2984:
2985:
2986:
2987:
2988:
2989:
2990:
2991:
2992:
2993:
2994:
2995:
2996:
2997:
2998:
2999:
3000:
3001:
3002:
3003:
3004:
3005:
3006:
3007:
3008:
3009:
3010:
3011:
3012:
3013:
3014:
3015:
3016:
3017:
3018:
3019:
3020:
3021:
3022:
3023:
3024:
3025:
3026:
3027:
3028:
3029:
3030:
3031:
3032:
3033:
3034:
3035:
3036:
3037:
3038:
3039:
3040:
3041:
3042:
3043:
3044:
3045:
3046:
3047:
3048:
3049:
3050:
3051:
3052:
3053:
3054:
3055:
3056:
3057:
3058:
3059:
3060:
3061:
3062:
3063:
3064:
3065:
3066:
3067:
3068:
3069:
3070:
3071:
3072:
3073:
3074:
3075:
3076:
3077:
3078:
3079:
3080:
3081:
3082:
3083:
3084:
3085:
3086:
3087:
3088:
3089:
3090:
3091:
3092:
3093:
3094:
3095:
3096:
3097:
3098:
3099:
3100:
3101:
3102:
3103:
3104:
3105:
3106:
3107:
3108:
3109:
3110:
3111:
3112:
3113:
3114:
3115:
3116:
3117:
3118:
3119:
3120:
3121:
3122:
3123:
3124:
3125:
3126:
3127:
3128:
3129:
3130:
3131:
3132:
3133:
3134:
3135:
3136:
3137:
3138:
3139:
3140:
3141:
3142:
3143:
3144:
3145:
3146:
3147:
3148:
3149:
3150:
3151:
3152:
3153:
3154:
3155:
3156:
3157:
3158:
3159:
3160:
3161:
3162:
3163:
3164:
3165:
3166:
3167:
3168:
3169:
3170:
3171:
3172:
3173:
3174:
3175:
3176:
3177:
3178:
3179:
3180:
3181:
3182:
3183:
3184:
3185:
3186:
3187:
3188:
3189:
3190:
3191:
3192:
3193:
3194:
3195:
3196:
3197:
3198:
3199:
3200:
3201:
3202:
3203:
3204:
3205:
3206:
3207:
3208:
3209:
3210:
3211:
3212:
3213:
3214:
3215:
3216:
3217:
3218:
3219:
3220:
3221:
3222:
3223:
3224:
3225:
3226:
3227:
3228:
3229:
3230:
3231:
3232:
3233:
3234:
3235:
3236:
3237:
3238:
3239:
3240:
3241:
3242:
3243:
3244:
3245:
3246:
3247:
3248:
3249:
3250:
3251:
3252:
3253:
3254:
3255:
3256:
3257:
3258:
3259:
3260:
3261:
3262:
3263:
3264:
3265:
3266:
3267:
3268:
3269:
3270:
3271:
3272:
3273:
3274:
3275:
3276:
3277:
3278:
3279:
3280:
3281:
3282:
3283:
3284:
3285:
3286:
3287:
3288:
3289:
3290:
3291:
3292:
3293:
3294:
3295:
3296:
3297:
3298:
3299:
3300:
3301:
3302:
3303:
3304:
3305:
3306:
3307:
3308:
3309:
3310:
3311:
3312:
3313:
3314:
3315:
3316:
3317:
3318:
3319:
3320:
3321:
3322:
3323:
3324:
3325:
3326:
3327:
3328:
3329:
3330:
3331:
3332:
3333:
3334:
3335:
3336:
3337:
3338:
3339:
3340:
3341:
3342:
3343:
3344:
3345:
3346:
3347:
3348:
3349:
3350:
3351:
3352:
3353:
3354:
3355:
3356:
3357:
3358:
3359:
3360:
3361:
3362:
3363:
3364:
3365:
3366:
3367:
3368:
3369:
3370:
3371:
3372:
3373:
3374:
3375:
3376:
3377:
3378:
3379:
3380:
3381:
3382:
3383:
3384:
3385:
3386:
3387:
3388:
3389:
3390:
3391:
3392:
3393:
3394:
3395:
3396:
3397:
3398:
3399:
3400:
3401:
3402:
3403:
3404:
3405:
3406:
3407:
3408:
3409:
3410:
3411:
3412:
3413:
3414:
3415:
3416:
3417:
3418:
3419:
3420:
3421:
3422:
3423:
3424:
3425:
3426:
3427:
3428:
3429:
3430:
3431:
3432:
3433:
3434:
3435:
3436:
3437:
3438:
3439:
3440:
3441:
3442:
3443:
3444:
3445:
3446:
3447:
3448:
3449:
3450:
3451:
3452:
3453:
3454:
3455:
3456:
3457:
3458:
3459:
3460:
3461:
3462:
3463:
3464:
3465:
3466:
3467:
3468:
3469:
3470:
3471:
3472:
3473:
3474:
3475:
3476:
3477:
3478:
3479:
3480:
3481:
3482:
3483:
3484:
3485:
3486:
3487:
3488:
3489:
3490:
3491:
3492:
3493:
3494:
3495:
3496:
3497:
3498:
3499:
3500:
3501:
3502:
3503:
3504:
3505:
3506:
3507:
3508:
3509:
3510:
3511:
3512:
3513:
3514:
3515:
3516:
3517:
3518:
3519:
3520:
3521:
3522:
3523:
3524:
3525:
3526:
3527:
3528:
3529:
3530:
3531:
3532:
3533:
3534:
3535:
3536:
3537:
3538:
3539:
3540:
3541:
3542:
3543:
3544:
3545:
3546:
3547:
3548:
3549:
3550:
3551:
3552:
3553:
3554:
3555:
3556:
3557:
3558:
3559:
3560:
3561:
3562:
3563:
3564:
3565:
3566:
3567:
3568:
3569:
3570:
3571:
3572:
3573:
3574:
3575:
3576:
3577:
3578:
3579:
3580:
3581:
3582:
3583:
3584:
3585:
3586:
3587:
3588:
3589:
3590:
3591:
3592:
3593:
3594:
3595:
3596:
3597:
3598:
3599:
3600:
3601:
3602:
3603:
3604:
3605:
3606:
3607:
3608:
3609:
3610:
3611:
3612:
3613:
3614:
3615:
3616:
3617:
3618:
3619:
3620:
3621:
3622:
3623:
3624:
3625:
3626:
3627:
3628:
3629:
3630:
3631:
3632:
3633:
3634:
3635:
3636:
3637:
3638:
3639:
3640:
3641:
3642:
3643:
3644:
3645:
3646:
3647:
3648:
3649:
3650:
3651:
3652:
3653:
3654:
3655:
3656:
3657:
3658:
3659:
3660:
3661:
3662:
3663:
3664:
3665:
3666:
3667:
3668:
3669:
3670:
3671:
3672:
3673:
3674:
3675:
3676:
3677:
3678:
3679:
3680:
3681:
3682:
3683:
3684:
3685:
3686:
3687:
3688:
3689:
3690:
3691:
3692:
3693:
3694:
3695:
3696:
3697:
3698:
3699:
3700:
3701:
3702:
3703:
3704:
3705:
3706:
3707:
3708:
3709:
3710:
3711:
3712:
3713:
3714:
3715:
3716:
3717:
3718:
3719:
3720:
3721:
3722:
3723:
3724:
3725:
3726:
3727:
3728:
3729:
3730:
3731:
3732:
3733:
3734:
3735:
3736:
3737:
3738:
3739:
3740:
3741:
3742:
3743:
3744:
3745:
3746:
3747:
3748:
3749:
3750:
3751:
3752:
3753:
3754:
3755:
3756:
3757:
3758:
3759:
3760:
3761:
3762:
3763:
3764:
3765:
3766:
3767:
3768:
3769:
3770:
3771:
3772:
3773:
3774:
3775:
3776:
3777:
3778:
3779:
3780:
3781:
3782:
3783:
3784:
3785:
3786:
3787:
3788:
3789:
3790:
3791:
3792:
3793:
3794:
3795:
3796:
3797:
3798:
3799:
3800:
3801:
3802:
3803:
3804:
3805:
3806:
3807:
3808:
3809:
3810:
3811:
3812:
3813:
3814:
3815:
3816:
3817:
3818:
3819:
3820:
3821:
3822:
3823:
3824:
3825:
3826:
3827:
3828:
3829:
3830:
3831:
3832:
3833:
3834:
3835:
3836:
3837:
3838:
3839:
3840:
3841:
3842:
3843:
3844:
3845:
3846:
3847:
3848:
3849:
3850:
3851:
3852:
3853:
3854:
3855:
3856:
3857:
3858:
3859:
3860:
3861:
3862:
3863:
3864:
3865:
3866:
3867:
3868:
3869:
3870:
3871:
3872:
3873:
3874:
3875:
3876:
3877:
3878:
3879:
3880:
3881:
3882:
3883:
3884:
3885:
3886:
3887:
3888:
3889:
3890:
3891:
3892:
3893:
3894:
3895:
3896:
3897:
3898:
3899:
3900:
3901:
3902:
3903:
3904:
3905:
3906:
3907:
3908:
3909:
3910:
3911:
3912:
3913:
3914:
3915:
3916:
3917:
3918:
3919:
3920:
3921:
3922:
3923:
3924:
3925:
3926:
3927:
3928:
3929:
3930:
3931:
3932:
3933:
3934:
3935:
3936:
3937:
3938:
3939:
3940:
3941:
3942:
3943:
3944:
3945:
3946:
3947:
3948:
3949:
3950:
3951:
3952:
3953:
3954:
3955:
3956:
3957:
3958:
3959:
3960:
3961:
3962:
3963:
3964:
3965:
3966:
3967:
3968:
3969:
3970:
3971:
3972:
3973:
3974:
3975:
3976:
3977:
3978:
3979:
3980:
3981:
3982:
3983:
3984:
3985:
3986:
3987:
3988:
3989:
3990:
3991:
3992:
3993:
3994:
3995:
3996:
3997:
3998:
3999:
4000:

```

```

2630:     php_error_docref(NULL, E_WARNING, "escape must be a character");
2631:     RETURN_FALSE;
2632: }
2633: escape = esc[0];
2634: /* no break */
2635: case 3:
2636:     if (strlen != 1) {
2637:         php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2638:         RETURN_FALSE;
2639:     }
2640:     enclosure = enclo[0];
2641:     /* no break */
2642: case 2:
2643:     if (strlen != 1) {
2644:         php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2645:         RETURN_FALSE;
2646:     }
2647:     delimiter = delim[0];
2648:     /* no break */
2649: case 1:
2650:     break;
2651: }
2652: ret = php_fputcsv(intern->u.file.stream, fields, delimiter, enclosure, escape);
2653: RETURN_LONG(ret);
2654: }
2655: }
2656: /* }}} */
2657:
2658: /* {{{ proto void SpFileObject::setCsvControl(string delimiter [, string enclosure [, string escape ]])
2659:  * Set the delimiter, enclosure and escape character used in fputcsv */
2660: SPL_METHOD(SpFileObject, setCsvControl)
2661: {
2662:     spl_filesystem_object *intern = Z_SPFLILESYSTEM_P(getThis());
2663:     char *delimiter = "", enclosure = "", escape = "";
2664:     char *delim = NULL, *enclo = NULL, *esc = NULL;
2665:     size_t delim_len = 0, enclo_len = 0, esc_len = 0;
2666:
2667:     if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "sss", &delim, &delim_len, &enclo, &enclo_len, &esc, &esc_len) == SUCCESS) {
2668:         switch(ZEND_NUM_ARGS())
2669:         {
2670:             case 3:
2671:                 if (esc_len != 1) {
2672:                     php_error_docref(NULL, E_WARNING, "escape must be a character");
2673:                     RETURN_FALSE;
2674:                 }
2675:                 escape = esc[0];
2676:                 /* no break */
2677:             case 2:
2678:                 if (delim_len != 1) {
2679:                     php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2680:                     RETURN_FALSE;
2681:                 }
2682:                 enclosure = enclo[0];
2683:                 /* no break */
2684:             case 1:
2685:                 if (delim_len != 1) {
2686:                     php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2687:                     RETURN_FALSE;
2688:                 }
2689:                 delimiter = delim[0];
2690:                 /* no break */
2691:             case 0:
2692:                 break;
2693:         }
2694:         intern->u.file.delimiter = delimiter;
2695:         intern->u.file.enclosure = enclosure;
2696:         intern->u.file.escape = escape;
2697:     }
2698:     /* }}} */
2699: }
2700:
2701: /* {{{ proto array SpFileObject::getCsvControl()
2702:  * Get the delimiter, enclosure and escape character used in fputcsv */
2703: SPL_METHOD(SpFileObject, getCsvControl)
2704: {
2705:     spl_filesystem_object *intern = Z_SPFLILESYSTEM_P(getThis());
2706:     char delimiter[2], enclosure[2], escape[2];
2707:
2708:     array_init(return_value);
2709:
2710:     delimiter[0] = intern->u.file.delimiter;
2711:     delimiter[1] = '\0';
2712:     enclosure[0] = intern->u.file.enclosure;
2713:     enclosure[1] = '\0';
2714:     escape[0] = intern->u.file.escape;
2715:     escape[1] = '\0';
2716:
2717:     add_next_index_string(return_value, delimiter);
2718:     add_next_index_string(return_value, enclosure);
2719:     add_next_index_string(return_value, escape);
2720: }
2721: /* }}} */
2722:
2723: /* {{{ proto bool SpFileObject::flock(int operation [, int wouldblock])
2724:  * Portable file locking */
2725: SPL_METHOD(SpFileObject, flock)
2726: {
2727:     /* {{{ proto bool SpFileObject::fflush()
2728:      * Flush the file */
2729:     SPL_METHOD(SpFileObject, fflush)
2730:     {
2731:         spl_filesystem_object *intern = Z_SPFLILESYSTEM_P(getThis());
2732:
2733:         if (intern->u.file.stream) {
2734:             zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialised");
2735:             return;
2736:         }
2737:
2738:         RETURN_BOOL(php_stream_flush(intern->u.file.stream));
2739:     }
2740:     /* }}} */
2741:
2742:     /* {{{ proto int SpFileObject::ftell()
2743:      * Return current file position */
2744:     SPL_METHOD(SpFileObject, ftell)
2745:     {
2746:         spl_filesystem_object *intern = Z_SPFLILESYSTEM_P(getThis());
2747:         zend_long ret;
2748:
2749:         if (intern->u.file.stream) {
2750:             zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialised");
2751:             return;
2752:         }
2753:
2754:         ret = php_stream_tell(intern->u.file.stream);
2755:
2756:         if (ret == -1) {
2757:             RETURN_FALSE;
2758:         }
2759:         RETURN_LONG(ret);
2760:     }
2761:     /* }}} */
2762:
2763:     /* {{{ proto int SpFileObject::fseek(int pos [, int whence = SEEK_SET])
2764:      * Return current file position */
2765:     SPL_METHOD(SpFileObject, fseek)
2766:     {
2767:         spl_filesystem_object *intern = Z_SPFLILESYSTEM_P(getThis());
2768:         zend_long pos, whence = SEEK_SET;
2769:
2770:         if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "li", &pos, &whence) == FAILURE) {
2771:             return;
2772:         }
2773:
2774:         if (intern->u.file.stream) {
2775:             zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialised");
2776:             return;
2777:         }
2778:
2779:         spl_filesystem_file_free_line(intern);
2780:         RETURN_LONG(php_stream_seek(intern->u.file.stream, pos, (int)whence));
2781:     }
2782:     /* }}} */
2783:
2784:     /* {{{ proto int SpFileObject::fgetc()
2785:      * Get a character from the file */
2786:     SPL_METHOD(SpFileObject, fgetc)
2787:     {
2788:         spl_filesystem_object *intern = Z_SPFLILESYSTEM_P(getThis());
2789:         char buf[2];
2790:         int result;
2791:
2792:         if (intern->u.file.stream) {
2793:             zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialised");
2794:             return;
2795:         }
2796:
2797:         spl_filesystem_file_free_line(intern);
2798:         result = php_stream_getc(intern->u.file.stream);
2799:
2800:         if (result == EOF) {
2801:             RETURN_FALSE;
2802:         }
2803:         if (result == '\n') {
2804:             intern->u.file.current_line_num++;
2805:         }
2806:         buf[0] = result;
2807:         buf[1] = '\0';
2808:
2809:         RETURN_STRING(buf, 1);
2810:     }
2811:     /* }}} */
2812:
2813:     /* {{{ proto string SpFileObject::fgets(string allowable_chars)
2814:      * Get a line from file pointer and strip BOM tags */
2815:     SPL_METHOD(SpFileObject, fgets)
2816:
2817:     spl_filesystem_object *intern = Z_SPFLILESYSTEM_P(getThis());
2818:     zend_long line_pos;
2819:     if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "l", &line_pos) == FAILURE) {
2820:         return;
2821:     }
2822:     spl_filesystem_file_free_line(intern);
2823:     result = php_stream_getc(intern->u.file.stream);
2824:     if (result == EOF) {
2825:         RETURN_FALSE;
2826:     }
2827:     if (result == '\n') {
2828:         intern->u.file.current_line_num++;
2829:     }
2830:     buf[0] = result;
2831:     buf[1] = '\0';
2832:     RETURN_STRING(buf, 1);
2833: }
2834:
2835: /* }}} */
2836:
2837: /* {{{ proto bool SpFileObject::ftruncate(int size)
2838:  * Truncate file to 'size' length */
2839: SPL_METHOD(SpFileObject, ftruncate)
2840: {
2841:     spl_filesystem_object *intern = Z_SPFLILESYSTEM_P(getThis());
2842:     zend_long size;
2843:
2844:     if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "l", &size) == FAILURE) {
2845:         return;
2846:     }
2847:
2848:     if (!php_stream_truncate_supported(intern->u.file.stream)) {
2849:         zend_throw_exception_ex(spl_ce_LogicException, 0, "Can't truncate file %s", intern->u.file_name);
2850:         RETURN_FALSE;
2851:     }
2852:
2853:     RETURN_BOOL(0 == php_stream_truncate_set_size(intern->u.file.stream, size));
2854:     /* }}} */
2855:
2856:     /* {{{ proto void SpFileObject::seek(int line_pos)
2857:      * Seek to specified line */
2858:     SPL_METHOD(SpFileObject, seek)
2859:     {
2860:         spl_filesystem_object *intern = Z_SPFLILESYSTEM_P(getThis());
2861:         zend_long line_pos;
2862:
2863:         if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "l", &line_pos) == FAILURE) {
2864:             return;
2865:         }
2866:
2867:         if (intern->u.file.stream) {
2868:             zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialised");
2869:             return;
2870:         }
2871:
2872:         if (line_pos < 0) {
2873:             zend_throw_exception_ex(spl_ce_LogicException, 0, "Can't seek file to negative line " . LONG_FMT, intern->u.file_name, line_pos);
2874:             RETURN_FALSE;
2875:         }
2876:
2877:         spl_filesystem_file_rewind(getThis(), intern);
2878:
2879:         while (intern->u.file.current_line_num < line_pos) {
2880:             if (spl_filesystem_file_read_line(getThis(), intern, 1) == FAILURE) {
2881:                 break;
2882:             }
2883:         }
2884:     }
2885:     /* }}} */
2886:
2887:     /* {{{ Function/Class/Method definitions */
2888:     ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_construct, 0, 0, 1)
2889:         ZEND_ARG_INFO(0, file_name)
2890:         ZEND_ARG_INFO(0, open_mode)
2891:         ZEND_ARG_INFO(0, use_include_path)
2892:         ZEND_ARG_INFO(0, context)
2893:     ZEND_END_ARG_INFO()
2894:
2895:     ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_setFlags, 0)
2896:         ZEND_ARG_INFO(0, flags)
2897:     ZEND_END_ARG_INFO()
2898:
2899:     ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_setMaxLineLen, 0)
2900:         ZEND_ARG_INFO(0, max_len)
2901:     ZEND_END_ARG_INFO()
2902:
2903:     ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fgetcsv, 0, 0, 0)
2904:         ZEND_ARG_INFO(0, delimiter)
2905:         ZEND_ARG_INFO(0, enclosure)
2906:         ZEND_ARG_INFO(0, escape)
2907:     ZEND_END_ARG_INFO()
2908:
2909:     ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fputcsv, 0, 0, 4)
2910:         ZEND_ARG_INFO(0, fields)
2911:         ZEND_ARG_INFO(0, delimiter)
2912:         ZEND_ARG_INFO(0, enclosure)
2913:         ZEND_ARG_INFO(0, escape)
2914:     ZEND_END_ARG_INFO()
2915:
2916:     ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_flock, 0, 0, 2)
2917:         ZEND_ARG_INFO(0, operation)
2918:         ZEND_ARG_INFO(0, wouldblock)
2919:     ZEND_END_ARG_INFO()
2920:
2921:     ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fflush, 0, 0, 0)
2922:     ZEND_END_ARG_INFO()
2923:
2924:     ZEND_BEGIN_ARG
```



```
3005: ZEND_ARG_INFO(0, escape)
3006: ZEND_END_ARG_INFO()
3007:
3008: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fputcsv, 0, 0, 1)
3009: ZEND_ARG_INFO(0, fields)
3010: ZEND_ARG_INFO(0, delimiter)
3011: ZEND_ARG_INFO(0, enclosure)
3012: ZEND_ARG_INFO(0, escape)
3013: ZEND_END_ARG_INFO()
3014:
3015: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_flock, 0, 0, 1)
3016: ZEND_ARG_INFO(0, operation)
3017: ZEND_ARG_INFO(1, wouldblock)
3018: ZEND_END_ARG_INFO()
3019:
3020: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fseek, 0, 0, 1)
3021: ZEND_ARG_INFO(0, pos)
3022: ZEND_ARG_INFO(0, whence)
3023: ZEND_END_ARG_INFO()
3024:
3025: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fgetas, 0, 0, 0)
3026: ZEND_ARG_INFO(0, allowable_tags)
3027: ZEND_END_ARG_INFO()
3028:
3029: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fscanf, 0, 0, 1)
3030: ZEND_ARG_INFO(0, format)
3031: ZEND_ARG_VARIADIC_INFO(1, vars)
3032: ZEND_END_ARG_INFO()
3033:
3034: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fwrite, 0, 0, 1)
3035: ZEND_ARG_INFO(0, str)
3036: ZEND_ARG_INFO(0, length)
3037: ZEND_END_ARG_INFO()
3038:
3039: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fread, 0, 0, 1)
3040: ZEND_ARG_INFO(0, length)
3041: ZEND_END_ARG_INFO()
3042:
3043: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_ftruncate, 0, 0, 1)
3044: ZEND_ARG_INFO(0, size)
3045: ZEND_END_ARG_INFO()
3046:
3047: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_seek, 0, 0, 1)
3048: ZEND_ARG_INFO(0, line_pos)
3049: ZEND_END_ARG_INFO()
3050:
3051: static const zend_function_entry spl_SplFileInfo_functions[] = {
3052:     SPL_ME(SplFileInfo, __construct, arginfo_file_object__construct, ZEND_ACC_PUBLIC)
3053:     SPL_ME(SplFileInfo, rewind, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3054:     SPL_ME(SplFileInfo, eof, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3055:     SPL_ME(SplFileInfo, valid, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3056:     SPL_ME(SplFileInfo, fgetc, arginfo_file_object_fgetc, ZEND_ACC_PUBLIC)
3057:     SPL_ME(SplFileInfo, fgetcsv, arginfo_file_object_fgetcsv, ZEND_ACC_PUBLIC)
3058:     SPL_ME(SplFileInfo, fputc, arginfo_file_object_fputc, ZEND_ACC_PUBLIC)
3059:     SPL_ME(SplFileInfo, setAccessControl, arginfo_file_object_fsetcsv, ZEND_ACC_PUBLIC)
3060:     SPL_ME(SplFileInfo, getAccessControl, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3061:     SPL_ME(SplFileInfo, flock, arginfo_file_object_flock, ZEND_ACC_PUBLIC)
3062:     SPL_ME(SplFileInfo, fflush, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3063:     SPL_ME(SplFileInfo, ftruncate, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3064:     SPL_ME(SplFileInfo, fseek, arginfo_file_object_fseek, ZEND_ACC_PUBLIC)
3065:     SPL_ME(SplFileInfo, fgetc, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3066:     SPL_ME(SplFileInfo, fpassthru, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3067:     SPL_ME(SplFileInfo, fgetas, arginfo_file_object_fgetas, ZEND_ACC_PUBLIC)
3068:     SPL_ME(SplFileInfo, fscanf, arginfo_file_object_fscanf, ZEND_ACC_PUBLIC)
3069:     SPL_ME(SplFileInfo, fwrite, arginfo_file_object_fwrite, ZEND_ACC_PUBLIC)
3070:     SPL_ME(SplFileInfo, fread, arginfo_file_object_fread, ZEND_ACC_PUBLIC)
3071:     SPL_ME(SplFileInfo, fstat, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3072:     SPL_ME(SplFileInfo, truncate, arginfo_file_object_ftruncate, ZEND_ACC_PUBLIC)
3073:     SPL_ME(SplFileInfo, current, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3074:     SPL_ME(SplFileInfo, key, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3075:     SPL_ME(SplFileInfo, next, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3076:     SPL_ME(SplFileInfo, setFlags, arginfo_file_object_setFlags, ZEND_ACC_PUBLIC)
3077:     SPL_ME(SplFileInfo, getFlags, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3078:     SPL_ME(SplFileInfo, setMaxLineLen, arginfo_file_object_setMaxLineLen, ZEND_ACC_PUBLIC)
3079:     SPL_ME(SplFileInfo, getMaxLineLen, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3080:     SPL_ME(SplFileInfo, getChilden, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3081:     SPL_ME(SplFileInfo, getChilden, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3082:     SPL_ME(SplFileInfo, seek, arginfo_file_object_seek, ZEND_ACC_PUBLIC)
3083:     /* mappings */
3084:     SPL_ME(SplFileInfo, getCurrentLine, SplFileInfo, fgetc, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3085:     SPL_ME(SplFileInfo, __toString, SplFileInfo, current, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3086:     PHP_FE_END
3087: };
3088:
3089: ZEND_BEGIN_ARG_INFO_EX(arginfo_temp_file_object__construct, 0, 0, 0)
3090: ZEND_ARG_INFO(0, max_memory)
3091: ZEND_END_ARG_INFO()
3092:
3093: static const zend_function_entry spl_SplTempFileObject_functions[] = {
3094:     SPL_ME(SplTempFileObject, __construct, arginfo_temp_file_object__construct, ZEND_ACC_PUBLIC)
3095:     PHP_FE_END
3096: };
3097: /* }}} */
3098:
3099: /* {{{ PHP_MINIT_FUNCTION(spl_directory)
3100: */
3101: #if PHP_MINIT_FUNCTION(spl_directory)
3102: {
3103:     REGISTER_SPL_STD_CLASS_EX(SplFileInfo, spl_filesystem_object_new, spl_SplFileInfo_functions);
3104:     memory(spl_filesystem_object_handlers, zend_get_object_handlers(), sizeof(zend_object_handlers));
3105:     spl_filesystem_object_handlers.offset = XOffsetOf(spl_filesystem_object, std);
3106:     spl_filesystem_object_handlers.clone_obj = spl_filesystem_object_clone;
3107:     spl_filesystem_object_handlers.cast_obj = spl_filesystem_object_cast;
3108:     spl_filesystem_object_handlers.get_debug_info = spl_filesystem_object_get_debug_info;
3109:     spl_filesystem_object_handlers.dtor_obj = spl_filesystem_object_destroy_object;
3110:     spl_filesystem_object_handlers.free_obj = spl_filesystem_object_free_storage;
3111:     spl_on_SplFileInfo->serialize = zend_class_serialize_deny;
3112:     spl_on_SplFileInfo->unserialize = zend_class_unserialize_deny;
3113:
3114:
3115:     REGISTER_SPL_SUB_CLASS_EX(DirectoryIterator, SplFileInfo, spl_filesystem_object_new, spl_DirectoryIterator_functions);
3116:     zend_class_implements(spl_on_DirectoryIterator, 1, zend_on_iterator);
3117:     REGISTER_SPL_IMPLMENTS(DirectoryIterator, SeekableIterator);
3118:
3119:     spl_on_DirectoryIterator->get_iterator = spl_filesystem_dir_get_iterator;
3120:
3121:     REGISTER_SPL_SUB_CLASS_EX(FilesystemIterator, DirectoryIterator, spl_filesystem_object_new, spl_filesystem_iterator_functions);
3122:
3123:     REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_MODE_MASK", SPL_FILE_DIR_CURRENT_MODE_MASK);
3124:     REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_AS_PATHNAME", SPL_FILE_DIR_CURRENT_AS_PATHNAME);
3125:     REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_AS_FILEINFO", SPL_FILE_DIR_CURRENT_AS_FILEINFO);
3126:     REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_AS_SELF", SPL_FILE_DIR_CURRENT_AS_SELF);
3127:     REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "KEY_MODE_MASK", SPL_FILE_DIR_KEY_MODE_MASK);
3128:     REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "KEY_AS_PATHNAME", SPL_FILE_DIR_KEY_AS_PATHNAME);
3129:     REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "FOLLOW_SYMLINKS", SPL_FILE_DIR_FOLLOW_SYMLINKS);
3130:     REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "KEY_AS_FILENAME", SPL_FILE_DIR_KEY_AS_FILENAME);
3131:     REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "NEW_CURRENT_AS_SELF", SPL_FILE_DIR_KEY_AS_FILENAME(SPL_FILE_DIR_CURRENT_AS_FILEINFO));
3132:     REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "OTHER_MODE_MASK", SPL_FILE_DIR_OTHERS_MASK);
3133:     REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "SKIP_DOTS", SPL_FILE_DIR_SKIPDOTS);
3134:     REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "UNIX_PATHS", SPL_FILE_DIR_UNIXPATHS);
3135:
3136:     spl_on_FilesystemIterator->get_iterator = spl_filesystem_tree_get_iterator;
3137:
3138:     REGISTER_SPL_SUB_CLASS_EX(RecursiveDirectoryIterator, FilesystemIterator, spl_filesystem_object_new, spl_RecursiveDirectoryIterator_functions);
3139:     REGISTER_SPL_IMPLMENTS(RecursiveDirectoryIterator, RecursiveIterator);
3140:
3141:     memory(spl_filesystem_object_check_handlers, spl_filesystem_object_handlers, sizeof(zend_object_handlers));
3142:     spl_filesystem_object_check_handlers.get_method = spl_filesystem_object_get_method_check;
3143:
3144: #ifdef HAVE_GLOB
3145:     REGISTER_SPL_SUB_CLASS_EX(GlobIterator, FilesystemIterator, spl_filesystem_object_new_check, spl_GlobIterator_functions);
3146:     REGISTER_SPL_IMPLMENTS(GlobIterator, Countable);
3147: #endif
3148:
3149:     REGISTER_SPL_SUB_CLASS_EX(SplFileInfo, SplFileInfo, spl_filesystem_object_new_check, spl_SplFileInfo_functions);
3150:     REGISTER_SPL_IMPLMENTS(SplFileInfo, RecursiveIterator);
3151:     REGISTER_SPL_IMPLMENTS(SplFileInfo, SeekableIterator);
3152:
3153:     REGISTER_SPL_CLASS_CONST_LONG(SplFileInfo, "DROP_NEW_LINK", SPL_FILE_OBJECT_DROP_NEW_LINK);
3154:     REGISTER_SPL_CLASS_CONST_LONG(SplFileInfo, "READ_AHEAD", SPL_FILE_OBJECT_READ_AHEAD);
3155:     REGISTER_SPL_CLASS_CONST_LONG(SplFileInfo, "READ_BUFFER", SPL_FILE_OBJECT_READ_BUFFER);
3156:     REGISTER_SPL_CLASS_CONST_LONG(SplFileInfo, "READ_CSV", SPL_FILE_OBJECT_READ_CSV);
3157:
3158:     REGISTER_SPL_SUB_CLASS_EX(SplTempFileObject, SplFileInfo, spl_filesystem_object_new_check, spl_SplTempFileObject_functions);
3159:     return SUCCESS;
3160: }
3161: /* }}} */
3162:
3163: /*
3164:  * Local variables:
3165:  * tab-width: 4
3166:  * c-basic-offset: 4
3167:  * End:
3168:  * vim600: noet sw=4 ts=4 fdm=marker
3169:  * vim600: noet sw=4 ts=4
3170:  */
```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | https://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_ITERATORS_H
22: #define SPL_ITERATORS_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26: #if HAVE_PCRE || HAVE_BUNDLED_PCRE
27: #include "ext/pcre/php_pcre.h"
28: #endif
29:
30: #define spl_ow_traversable send_ow_traversable
31: #define spl_ow_iterator send_ow_iterator
32: #define spl_ow_aggregate send_ow_aggregate
33: #define spl_ow_arrayaccess send_ow_arrayaccess
34: #define spl_ow_serializable send_ow_serializable
35: #define spl_ow_countable send_ow_countable
36:
37: #define PHPAPI zend_class_entry *spl_ow_recursiveiterator;
38: #define PHPAPI zend_class_entry *spl_ow_recursiveiteratoriterator;
39: #define PHPAPI zend_class_entry *spl_ow_recursiveiteratoriterator;
40: #define PHPAPI zend_class_entry *spl_ow_filteriterator;
41: #define PHPAPI zend_class_entry *spl_ow_recursivefilteriterator;
42: #define PHPAPI zend_class_entry *spl_ow_parentiterator;
43: #define PHPAPI zend_class_entry *spl_ow_weakableiterator;
44: #define PHPAPI zend_class_entry *spl_ow_limititerator;
45: #define PHPAPI zend_class_entry *spl_ow_cachingiterator;
46: #define PHPAPI zend_class_entry *spl_ow_recursivecachingiterator;
47: #define PHPAPI zend_class_entry *spl_ow_outertiterator;
48: #define PHPAPI zend_class_entry *spl_ow_iteratoriterator;
49: #define PHPAPI zend_class_entry *spl_ow_mohanditerator;
50: #define PHPAPI zend_class_entry *spl_ow_infiniteiterator;
51: #define PHPAPI zend_class_entry *spl_ow_emptyiterator;
52: #define PHPAPI zend_class_entry *spl_ow_appenditerator;
53: #define PHPAPI zend_class_entry *spl_ow_requeiterator;
54: #define PHPAPI zend_class_entry *spl_ow_recursiverequeiterator;
55: #define PHPAPI zend_class_entry *spl_ow_callbackfilteriterator;
56: #define PHPAPI zend_class_entry *spl_ow_recursivecallbackfilteriterator;
57:
58: #define PHP_MINIT_FUNCTION(spl_iterators);
59:
60: #define PHP_FUNCTION(iterator_to_array);
61: #define PHP_FUNCTION(iterator_count);
62: #define PHP_FUNCTION(iterator_apply);
63:
64: typedef enum {
65:     DIT_Default = 0,
66:     DIT_Filteriterator = DIT_Default,
67:     DIT_RecursiveFilteriterator = DIT_Default,
68:     DIT_Parentiterator = DIT_Default,
69:     DIT_Limititerator,
70:     DIT_Cachingiterator,
71:     DIT_RecursiveCachingiterator,
72:     DIT_Iteratoriterator,
73:     DIT_Mohanditerator,
74:     DIT_Infiniteiterator,
75:     DIT_Appenditerator,
76: #if HAVE_PCRE || HAVE_BUNDLED_PCRE
77:     DIT_Requeiterator,
78:     DIT_RecursiveRequeiterator,
79: #endif
80:     DIT_CallbackFilteriterator,
81:     DIT_RecursiveCallbackFilteriterator,
82:     DIT_Unknown = -1,
83: } dual_it_type;
84:
85: typedef enum {
86:     RIT_Default = 0,
87:     RIT_RecursiveIteratoriterator = RIT_Default,
88:     RIT_RecursiveTreeIterator,
89:     RIT_Unknown = -1,
90: } recursive_it_type;
91:
92: enum {
93:     /* public */
94:     CIT_CALL_POSTING = 0x00000001,
95:     CIT_YOSTRING_USER_KEY = 0x00000002,
96:     CIT_YOSTRING_USER_CURRENT = 0x00000004,
97:     CIT_YOSTRING_USER_INNER = 0x00000008,
98:     CIT_CATCH_GET_CHILD = 0x00000010,
99:     CIT_FULL_CACHE = 0x00000100,
100:     CIT_PUBLIC = 0x000000FFFF,
101:     /* private */
102:     CIT_VALID = 0x00010000,
103:     CIT_HAS_CHILDREN = 0x00020000
104: };
105:
106: enum {
107:     /* public */
108:     REGIT_USER_KEY = 0x00000001,
109:     REGIT_INVERTED = 0x00000002
110: };
111:
112: typedef enum {
113:     REGIT_MODE_MATCH,
114:     REGIT_MODE_GET_MATCH,
115:     REGIT_MODE_ALL_MATCHES,
116:     REGIT_MODE_SPLIT,
117:     REGIT_MODE_REPLACE,
118:     REGIT_MODE_MAX
119: } regex_mode;
120:
121: typedef struct _spl_chfilter_it_intern {
122:     zend_fcall_info fci;
123:     zend_fcall_info_cache fcci;
124:     zend_object *obj;
125: } _spl_chfilter_it_intern;
126:
127: typedef struct _spl_dual_it_object {
128:     struct {
129:         zval zobj;
130:         zend_class_entry *ce;
131:         zend_object *obj;
132:         zend_object_iterator *iterator;
133:     } inner;
134:     struct {
135:         zval data;
136:         zval key;
137:         zend_long pos;
138:     } current;
139:     dual_it_type dit_type;
140:     union {
141:         struct {
142:             zend_long offset;
143:             zend_long count;
144:         } limit;
145:         struct {
146:             zend_long flags; /* CIT_* */
147:             zval src;
148:             zval children;
149:             zval scache;
150:         } caching;
151:         struct {
152:             zval array;
153:             zend_object_iterator *iterator;
154:         } append;
155: #if HAVE_PCRE || HAVE_BUNDLED_PCRE
156:         struct {
157:             zend_long flags;
158:             zend_long preg_flags;
159:             pcre_cache_entry *pcre;
160:             zend_string *regex;
161:             regex_mode mode;
162:             int use_flags;
163:         } regex;
164: #endif
165:     };
166:     _spl_chfilter_it_intern *chfilter;
167: } _spl_dual_it_object;
168:
169: static inline _spl_dual_it_object *spl_dual_it_from_obj(zend_object *obj) {
170:     return (_spl_dual_it_object *) ((char *) (obj) - XOFFSETOF(_spl_dual_it_object, std));
171: }
172:
173: #define SPL_DUAL_IT_P(xv) spl_dual_it_from_obj((zend_object *) (xv))
174:
175: typedef int (*spl_iterator_apply_func_t)(zend_object_iterator *iter, void *puser);
176:
177: #define PHPAPI int spl_iterator_apply(zval *obj, spl_iterator_apply_func_t apply_func, void *puser);
178:
179: #endif /* SPL_ITERATORS_H */
180:
181:
182: /*
183:  * Local Variables:
184:  * * tab-width: 4
185:  * * tab-width: 4
186:  * * End:
187:  * * vim600: fdm=marker
188:  * * vim: noet sw=4 ts=4
189:  */
```

189: \*/



```

1: 1:  PHP Version 7.0.0
2: 2:  Copyright (c) 1997-2018 The PHP Group
3: 3:  This source file is subject to version 3.01 of the PHP license,
4: 4:  that is bundled with this package in the file LICENSE, and is
5: 5:  available through the world-wide-web at the following url:
6: 6:  http://www.php.net/license/3_01.txt
7: 7:  If you did not receive a copy of the PHP license and are unable to
8: 8:  obtain it through the world-wide-web, please send a note to
9: 9:  license@php.net so we can mail you a copy immediately.
10: 10:  Authors: Marcus Ruescher <chelly@php.net>
11: 11:
12: 12:
13: 13:
14: 14:
15: 15:
16: 16:
17: 17:
18: 18:
19: 19:
20: 20:
21: 21: #ifdef HAVE_CONFIG_H
22: 22: #include "config.h"
23: 23: #endif
24: 24:
25: 25: #include "php.h"
26: 26: #include "php_ini.h"
27: 27: #include "ext/standard/info.h"
28: 28: #include "ext/standard/php_var.h"
29: 29: #include "zend_smart_str.h"
30: 30: #include "zend_interfaces.h"
31: 31: #include "zend_exceptions.h"
32: 32:
33: 33: #include "php_api.h"
34: 34: #include "api_functions.h"
35: 35: #include "api_globals.h"
36: 36: #include "api_iterators.h"
37: 37: #include "api_array.h"
38: 38: #include "api_exceptions.h"
39: 39:
40: 40: zend_object_handlers spl_handler_ArrayObject;
41: 41: PHPAPI zend_class_entry *spl_array_array_iterator;
42: 42:
43: 43: zend_object_handlers spl_handler_ArrayIterator;
44: 44: PHPAPI zend_class_entry *spl_array_array_iterator;
45: 45: PHPAPI zend_class_entry *spl_array_recurse_array_iterator;
46: 46:
47: 47: #define SPL_ARRAY_STD_PROPS_LIST 0x00000001
48: 48: #define SPL_ARRAY_ARRAY_AS_PROPS 0x00000002
49: 49: #define SPL_ARRAY_CHILD_ARRAYS_ONLY 0x00000004
50: 50: #define SPL_ARRAY_OVERLOADED_NEXT 0x00010000
51: 51: #define SPL_ARRAY_OVERLOADED_VALID 0x00020000
52: 52: #define SPL_ARRAY_OVERLOADED_KEY 0x00040000
53: 53: #define SPL_ARRAY_OVERLOADED_CURRENT 0x00080000
54: 54: #define SPL_ARRAY_OVERLOADED_NEXT 0x00100000
55: 55: #define SPL_ARRAY_IS_SELF 0x01000000
56: 56: #define SPL_ARRAY_USE_OTHER 0x02000000
57: 57: #define SPL_ARRAY_INT_MASK 0xFF000000
58: 58: #define SPL_ARRAY_CLONE_MASK 0x100FFFFF
59: 59:
60: 60: #define SPL_ARRAY_METHOD_NO_ARG 0
61: 61: #define SPL_ARRAY_METHOD_USE_ARG 1
62: 62: #define SPL_ARRAY_METHOD_USE_ARG 2
63: 63:
64: 64: typedef struct _spl_array_object {
65: 65:     zval array;
66: 66:     uint32_t ht_iter;
67: 67:     int flags;
68: 68:     unsigned char *obj_properties;
69: 69:     zend_function *fptr_offset_get;
70: 70:     zend_function *fptr_offset_set;
71: 71:     zend_function *fptr_offset_key;
72: 72:     zend_function *fptr_offset_del;
73: 73:     zend_function *fptr_count;
74: 74:     zend_class_entry *ce_get_iterator;
75: 75:     zend_object std;
76: 76:     spl_array_object;
77: 77:
78: 78:     static inline spl_array_object *spl_array_from_obj(zend_object *obj) { /* {{{ */
79: 79:         return (spl_array_object *) (char *) (obj) - XOffsetOf(spl_array_object, std);
80: 80:     }
81: 81:     /* }}} */
82: 82:
83: 83: #define Z_SPARRAY_P(rv) spl_array_from_obj(Z_OBJ_P((rv)))
84: 84:
85: 85:     static inline HashTable **spl_array_get_hash_table_ptr(spl_array_object *intern) { /* {{{ */
86: 86:         /*??? TODO: Delay duplication for arrays; only duplicate for write operations
87: 87:         if (intern->var_flags & SPL_ARRAY_IS_SELF) {
88: 88:             if (intern->std.properties) {
89: 89:                 rebuild_obj_properties(intern->std);
90: 90:             }
91: 91:             return intern->std.properties;
92: 92:         } else if (intern->var_flags & SPL_ARRAY_USE_OTHER) {
93: 93:             spl_array_object *other = Z_SPARRAY_P(intern->array);
94: 94:             return spl_array_get_hash_table_ptr(other);
95: 95:         } else if (Z_TYPE(intern->array) == IS_ARRAY) {
96: 96:             return Z_ARRVAL(intern->array);
97: 97:         } else {
98: 98:             zend_object *obj = Z_OBJ(intern->array);
99: 99:             if (obj->properties) {
100: 100:                 rebuild_obj_properties(obj);
101: 101:             } else if (GC_REFCOUNT(obj->properties) > 1) {
102: 102:                 IF EXPECTED (! GC_FLAGS(obj->properties) & IS_ARRAY_IMMUTABLE);
103: 103:                 GC_REFCOUNT(obj->properties);
104: 104:             }
105: 105:             obj->properties = zend_array_dup(obj->properties);
106: 106:             return obj->properties;
107: 107:         }
108: 108:     }
109: 109:     /* }}} */
110: 110:     /* }}} */
111: 111:
112: 112:     static inline HashTable *spl_array_get_hash_table(spl_array_object *intern) { /* {{{ */
113: 113:         return *spl_array_get_hash_table_ptr(intern);
114: 114:     }
115: 115:     /* }}} */
116: 116:
117: 117:     static inline void spl_array_replace_hash_table(spl_array_object *intern, HashTable *ht) { /* {{{ */
118: 118:         HashTable **ht_ptr = spl_array_get_hash_table_ptr(intern);
119: 119:         zend_array_destroy(*ht_ptr);
120: 120:         *ht_ptr = ht;
121: 121:     }
122: 122:     /* }}} */
123: 123:
124: 124:     static inline zend_bool spl_array_is_object(spl_array_object *intern) { /* {{{ */
125: 125:         while (intern->var_flags & SPL_ARRAY_USE_OTHER) {
126: 126:             intern = Z_SPARRAY_P(intern->array);
127: 127:         }
128: 128:         return (intern->var_flags & SPL_ARRAY_IS_SELF) || (Z_TYPE(intern->array) == IS_OBJECT);
129: 129:     }
130: 130:     /* }}} */
131: 131:
132: 132:     static inline spl_array_skip_protected(spl_array_object *intern, HashTable *ht);
133: 133:
134: 134:     static inline void spl_array_create_ht_iter(HashTable *ht, spl_array_object *intern) { /* {{{ */
135: 135:         intern->ht_iter = zend_hash_iterator_add(ht, ht->internalPointer);
136: 136:         zend_hash_internal_pointer_reset_ex(ht, &ht->internalPointer);
137: 137:         spl_array_skip_protected(intern, ht);
138: 138:     }
139: 139:     /* }}} */
140: 140:
141: 141:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
142: 142:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
143: 143:             spl_array_create_ht_iter(ht, intern);
144: 144:         }
145: 145:         return &ht->internalPointer;
146: 146:     }
147: 147:     /* }}} */
148: 148:
149: 149:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
150: 150:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
151: 151:             spl_array_create_ht_iter(ht, intern);
152: 152:         }
153: 153:         return &ht->internalPointer;
154: 154:     }
155: 155:     /* }}} */
156: 156:
157: 157:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
158: 158:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
159: 159:             spl_array_create_ht_iter(ht, intern);
160: 160:         }
161: 161:         return &ht->internalPointer;
162: 162:     }
163: 163:     /* }}} */
164: 164:
165: 165:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
166: 166:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
167: 167:             spl_array_create_ht_iter(ht, intern);
168: 168:         }
169: 169:         return &ht->internalPointer;
170: 170:     }
171: 171:     /* }}} */
172: 172:
173: 173:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
174: 174:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
175: 175:             spl_array_create_ht_iter(ht, intern);
176: 176:         }
177: 177:         return &ht->internalPointer;
178: 178:     }
179: 179:     /* }}} */
180: 180:
181: 181:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
182: 182:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
183: 183:             spl_array_create_ht_iter(ht, intern);
184: 184:         }
185: 185:         return &ht->internalPointer;
186: 186:     }
187: 187:     /* }}} */
188: 188:
189: 189:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
190: 190:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
191: 191:             spl_array_create_ht_iter(ht, intern);
192: 192:         }
193: 193:         return &ht->internalPointer;
194: 194:     }
195: 195:     /* }}} */
196: 196:
197: 197:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
198: 198:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
199: 199:             spl_array_create_ht_iter(ht, intern);
200: 200:         }
201: 201:         return &ht->internalPointer;
202: 202:     }
203: 203:     /* }}} */
204: 204:
205: 205:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
206: 206:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
207: 207:             spl_array_create_ht_iter(ht, intern);
208: 208:         }
209: 209:         return &ht->internalPointer;
210: 210:     }
211: 211:     /* }}} */
212: 212:
213: 213:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
214: 214:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
215: 215:             spl_array_create_ht_iter(ht, intern);
216: 216:         }
217: 217:         return &ht->internalPointer;
218: 218:     }
219: 219:     /* }}} */
220: 220:
221: 221:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
222: 222:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
223: 223:             spl_array_create_ht_iter(ht, intern);
224: 224:         }
22
```

```

377:         case BP_VAR_RH:
378:             zend_error(E_NOTICE, "Undefined offset: " ZEND_LONG_FMT, index);
379:         case BP_VAR_W: {
380:             zval *value;
381:             ZVAL_UNDEF(value);
382:             retval = zend_hash_index_update(ht, index, value);
383:         }
384:     }
385: }
386: return retval;
387: case IS_REFERENCE:
388:     ZVAL_DEREF(offset);
389:     goto try_again;
390: default:
391:     zend_error(E_WARNING, "Illegal offset type");
392:     return (type == BP_VAR_P || type == BP_VAR_RH) ?
393:         &EG(uninitialized_val) : &EG(uninitialized_val);
394: }
395: /* }}} */
396:
397: static int spl_array_has_dimension(zval *object, zval *offset, int check_empty);
398:
399: static zval *spl_array_read_dimension_ex(int check_inherited, zval *object, zval *offset, int type, zval *rv) /* {{{ */
400: {
401:     spl_array_object *intern = Z_SPLARRAY_P(object);
402:     zval *ret;
403:
404:     if (check_inherited &&
405:         (intern->fptr_offset_get || (type == BP_VAR_IS && intern->fptr_offset_has))) {
406:         if (type == BP_VAR_IS) {
407:             if (!spl_array_has_dimension(object, offset, 0)) {
408:                 return &EG(uninitialized_val);
409:             }
410:         }
411:
412:         if (intern->fptr_offset_get) {
413:             zval tmp;
414:             if (!offset) {
415:                 ZVAL_UNDEF(&tmp);
416:                 offset = &tmp;
417:             } else {
418:                 SEPARATE_ARG_1_PTR(offset);
419:             }
420:
421:             zend_call_method_with_1_params(object, Z_OBJCE_P(object), &intern->fptr_offset_get, "offsetGet", rv, offset);
422:             zval_ptr_dtor(offset);
423:
424:             if (!IS_UNDEF_P(rv)) {
425:                 return rv;
426:             }
427:             return &EG(uninitialized_val);
428:         }
429:
430:         ret = spl_array_get_dimension_ptr(check_inherited, intern, offset, type);
431:
432:         /* When in a write context,
433:          * ZE has to be fooled into thinking this is in a reference set
434:          * by separating (if necessary) and returning as IS_REFERENCE (with refcount == 1)
435:          */
436:
437:         if (type == BP_VAR_W || type == BP_VAR_RH || type == BP_VAR_UNSET) &&
438:             IS_UNDEF_P(ret) &&
439:             EXPECTED(ret != &EG(uninitialized_val)) {
440:             ZVAL_UNDEF(ret, ret);
441:         }
442:
443:         return ret;
444:     } /* }}} */
445:
446: static zval *spl_array_read_dimension(zval *object, zval *offset, int type, zval *rv) /* {{{ */
447: {
448:     return spl_array_read_dimension_ex(1, object, offset, type, rv);
449: } /* }}} */
450:
451: static void spl_array_write_dimension_ex(int check_inherited, zval *object, zval *offset, zval *value) /* {{{ */
452: {
453:     spl_array_object *intern = Z_SPLARRAY_P(object);
454:     zend_long index;
455:     HashTable *ht;
456:
457:     if (check_inherited && intern->fptr_offset_set) {
458:         zval tmp;
459:
460:         if (!offset) {
461:             ZVAL_NULL(&tmp);
462:             offset = &tmp;
463:         } else {
464:             SEPARATE_ARG_1_PTR(offset);
465:         }
466:         zend_call_method_with_2_params(object, Z_OBJCE_P(object), &intern->fptr_offset_set, "offsetSet", NULL, offset, value);
467:         zval_ptr_dtor(offset);
468:         return;
469:     }
470:
471:     if (intern->objApplyCount > 0) {
472:         zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
473:         return;
474:     }
475:
476:     Z_TRY_ADDREF_P(value);
477:     if (!offset) {
478:         ht = spl_array_get_hash_table(intern);
479:         zend_hash_next_index_insert(ht, value);
480:         return;
481:     }
482:
483:     try_again:
484:     switch (Z_TYPE_P(offset)) {
485:         case IS_STRING:
486:             ht = spl_array_get_hash_table(intern);
487:             zend_symtable_update_and(ht, Z_STR_P(offset), value);
488:             return;
489:         case IS_DOUBLE:
490:             index = (zend_long)Z_DVAL_P(offset);
491:             goto num_index;
492:         case IS_RESOURCE:
493:             index = Z_RES_HANDLE_P(offset);
494:             goto num_index;
495:         case IS_FALSE:
496:             index = 0;
497:             goto num_index;
498:         case IS_TRUE:
499:             index = 1;
500:             goto num_index;
501:         case IS_LONG:
502:             index = Z_LVAL_P(offset);
503:             goto num_index;
504:         case IS_NULL:
505:             ht = spl_array_get_hash_table(intern);
506:             zend_hash_index_update(ht, index, value);
507:             return;
508:         case IS_REFERENCE:
509:             ht = spl_array_get_hash_table(intern);
510:             zend_hash_next_index_insert(ht, value);
511:             return;
512:         case IS_ARRAY:
513:             goto try_again;
514:         default:
515:             zend_error(E_WARNING, "Illegal offset type");
516:             zval_ptr_dtor(value);
517:             return;
518:     }
519: } /* }}} */
520:
521: static void spl_array_write_dimension(zval *object, zval *offset, zval *value) /* {{{ */
522: {
523:     spl_array_write_dimension_ex(1, object, offset, value);
524: } /* }}} */
525:
526: static void spl_array_unset_dimension_ex(int check_inherited, zval *object, zval *offset) /* {{{ */
527: {
528:     zend_long index;
529:     HashTable *ht;
530:     spl_array_object *intern = Z_SPLARRAY_P(object);
531:
532:     if (check_inherited && intern->fptr_offset_del) {
533:         SEPARATE_ARG_1_PTR(offset);
534:         zend_call_method_with_1_params(object, Z_OBJCE_P(object), &intern->fptr_offset_del, "offsetUnset", NULL, offset);
535:         zval_ptr_dtor(offset);
536:         return;
537:     }
538:
539:     if (intern->objApplyCount > 0) {
540:         zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
541:         return;
542:     }
543:
544:     try_again:
545:     switch (Z_TYPE_P(offset)) {
546:         case IS_STRING:
547:             ht = spl_array_get_hash_table(intern);
548:             if (ht == &EG(symtbl_table)) {
549:                 if (zend_delete_global_variable(Z_STR_P(offset))) {
550:                     zend_error(E_NOTICE, "Undefined index: %s", Z_STRVAL_P(offset));
551:                 }
552:             } else {
553:                 zval *data = zend_symtable_find(ht, Z_STR_P(offset));
554:
555:                 if (data) {
556:                     if (Z_TYPE_P(data) == IS_INDIRECT) {
557:                         data = Z_INDIRECT_P(data);
558:                     }
559:                     if (IS_UNDEF_P(data) == IS_UNDEF) {
560:                         zend_error(E_NOTICE, "Undefined index: %s", Z_STRVAL_P(offset));
561:                     }
562:                     ZVAL_UNDEF(data);
563:                     zend_hash_move_forward_ex(ht, spl_array_get_pos_ptr(ht, intern));
564:                 }
565:             }
566:         case IS_DOUBLE:
567:             index = (zend_long)Z_DVAL_P(offset);
568:             goto num_index;
569:         case IS_RESOURCE:
570:             index = Z_RES_HANDLE_P(offset);
571:             goto num_index;
572:         case IS_FALSE:
573:             index = 0;
574:             goto num_index;
575:         case IS_TRUE:
576:             index = 1;
577:             goto num_index;
578:         case IS_LONG:
579:             index = Z_LVAL_P(offset);
580:             goto num_index;
581:         case IS_NULL:
582:             ht = spl_array_get_hash_table(intern);
583:             if (zend_hash_index_del(ht, index) == FAILURE) {
584:                 zend_error(E_NOTICE, "Undefined offset: " ZEND_LONG_FMT, index);
585:             }
586:             break;
587:         case IS_REFERENCE:
588:             ZVAL_DEREF(offset);
589:             goto try_again;
590:         default:
591:             zend_error(E_WARNING, "Illegal offset type");
592:             return;
593:     }
594:
595:     if (intern->fptr_offset_get) {
596:         value = spl_array_read_dimension_ex(1, object, offset, BP_VAR_R, rv);
597:     } else {
598:         value = tmp;
599:     }
600:
601:     zend_bool result = check_empty ? zend_is_true(value) : Z_TYPE_P(value) != IS_NULL;
602:     if (value == &rv) {
603:         zval_ptr_dtor(rv);
604:     }
605:     return result;
606: } /* }}} */
607:
608: static int spl_array_has_dimension(zval *object, zval *offset, int check_empty) /* {{{ */
609: {
610:     return spl_array_has_dimension_ex(1, object, offset, check_empty);
611: } /* }}} */
612:
613: static inline int spl_array_object_verify_pos_ex(spl_array_object *object, HashTable *ht, const char *msg_prefix)
614: {
615:     if (!ht) {
616:         zend_error_docref(NULL, E_NOTICE, "%sarray was modified outside object and is no longer an array", msg_prefix);
617:         return FAILURE;
618:     }
619:     return SUCCESS;
620: } /* }}} */
621:
622: static inline int spl_array_object_verify_pos(spl_array_object *object, HashTable *ht)
623: {
624:     return spl_array_object_verify_pos_ex(object, ht, "");
625: } /* }}} */
626:
627: /* {{{ proto bool ArrayObject::offsetExists(mixed $index)
628:    Returns whether the requested $index exists. */
629: SPL_METHOD(Array, offsetExists)
630: {
631:     zval *index;
632:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
633:         return;
634:     }
635:     return zend_bool(spl_array_has_dimension_ex(0, getThis(), index, 2));
636: } /* }}} */
637:
638: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
639:    Returns the value at the specified $index. */
640: SPL_METHOD(Array, offsetGet)
641: {
642:     zval *value, *index;
643:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
644:         return;
645:     }
646:     value = spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
647:     ZVAL_COPY(return_value, value);
648: } /* }}} */
649:
650: /* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $value)
651:    Sets the value at the specified $index to $value. */
652: SPL_METHOD(Array, offsetSet)
653: {
654:     zval *index, *value;
655:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "aa", &index, &value) == FAILURE) {
656:         return;
657:     }
658:     spl_array_write_dimension_ex(1, getThis(), index, value);
659: } /* }}} */
660:
661: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
662:    Unsets the value at the specified $index. */
663: SPL_METHOD(Array, offsetUnset)
664: {
665:     zval *index;
666:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
667:         return;
668:     }
669:     spl_array_unset_dimension_ex(1, getThis(), index);
670: } /* }}} */
671:
672: /* {{{ proto mixed ArrayObject::offsetIsset(mixed $index)
673:    Returns whether the requested $index exists. */
674: SPL_METHOD(Array, offsetIsset)
675: {
676:     zval *index;
677:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
678:         return;
679:     }
680:     return spl_array_has_dimension_ex(0, getThis(), index, 2);
681: } /* }}} */
682:
683: /* {{{ proto mixed ArrayObject::offsetUnset(mixed $index)
684:    Unsets the value at the specified $index. */
685: SPL_METHOD(Array, offsetUnset)
686: {
687:     zval *index;
688:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
689:         return;
690:     }
691:     spl_array_unset_dimension_ex(1, getThis(), index);
692: } /* }}} */
693:
694: /* {{{ proto mixed ArrayObject::offsetIsset(mixed $index)
695:    Returns whether the requested $index exists. */
696: SPL_METHOD(Array, offsetIsset)
697: {
698:     zval *index;
699:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
700:         return;
701:     }
702:     return spl_array_has_dimension_ex(0, getThis(), index, 2);
703: } /* }}} */
704:
705: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
706:    Returns the value at the specified $index. */
707: SPL_METHOD(Array, offsetGet)
708: {
709:     zval *index;
710:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
711:         return;
712:     }
713:     return spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
714: } /* }}} */
715:
716: /* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $value)
717:    Sets the value at the specified $index to $value. */
718: SPL_METHOD(Array, offsetSet)
719: {
720:     zval *index, *value;
721:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "aa", &index, &value) == FAILURE) {
722:         return;
723:     }
724:     spl_array_write_dimension_ex(1, getThis(), index, value);
725: } /* }}} */
726:
727: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
728:    Unsets the value at the specified $index. */
729: SPL_METHOD(Array, offsetUnset)
730: {
731:     zval *index;
732:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
733:         return;
734:     }
735:     spl_array_unset_dimension_ex(1, getThis(), index);
736: } /* }}} */
737:
738: /* {{{ proto mixed ArrayObject::offsetIsset(mixed $index)
739:    Returns whether the requested $index exists. */
740: SPL_METHOD(Array, offsetIsset)
741: {
742:     zval *index;
743:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
744:         return;
745:     }
746:     return spl_array_has_dimension_ex(0, getThis(), index, 2);
747: } /* }}} */
748:
749: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
750:    Returns the value at the specified $index. */
751: SPL_METHOD(Array, offsetGet)
752: {
753:     zval *index;
754:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
755:         return;
756:     }
757:     return spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
758: } /* }}} */
759:
760: /* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $value)
761:    Sets the value at the specified $index to $value. */
762: SPL_METHOD(Array, offsetSet)
763: {
764:     zval *index, *value;
765:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "aa", &index, &value) == FAILURE) {
766:         return;
767:     }
768:     spl_array_write_dimension_ex(1, getThis(), index, value);
769: } /* }}} */
770:
771: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
772:    Unsets the value at the specified $index. */
773: SPL_METHOD(Array, offsetUnset)
774: {
775:     zval *index;
776:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
777:         return;
778:     }
779:     spl_array_unset_dimension_ex(1, getThis(), index);
780: } /* }}} */
781:
782: /* {{{ proto mixed ArrayObject::offsetIsset(mixed $index)
783:    Returns whether the requested $index exists. */
784: SPL_METHOD(Array, offsetIsset)
785: {
786:     zval *index;
787:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
788:         return;
789:     }
790:     return spl_array_has_dimension_ex(0, getThis(), index, 2);
791: } /* }}} */
792:
793: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
794:    Returns the value at the specified $index. */
795: SPL_METHOD(Array, offsetGet)
796: {
797:     zval *index;
798:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
799:         return;
800:     }
801:     return spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
802: } /* }}} */
803:
804: /* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $value)
805:    Sets the value at the specified $index to $value. */
806: SPL_METHOD(Array, offsetSet)
807: {
808:     zval *index, *value;
809:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "aa", &index, &value) == FAILURE) {
810:         return;
811:     }
812:     spl_array_write_dimension_ex(1, getThis(), index, value);
813: } /* }}} */
814:
815: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
816:    Unsets the value at the specified $index. */
817: SPL_METHOD(Array, offsetUnset)
818: {
819:     zval *index;
820:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
821:         return;
822:     }
823:     spl_array_unset_dimension_ex(1, getThis(), index);
824: } /* }}} */
825:
826: /* {{{ proto mixed ArrayObject::offsetIsset(mixed $index)
827:    Returns whether the requested $index exists. */
828: SPL_METHOD(Array, offsetIsset)
829: {
830:     zval *index;
831:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
832:         return;
833:     }
834:     return spl_array_has_dimension_ex(0, getThis(), index, 2);
835: } /* }}} */
836:
837: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
838:    Returns the value at the specified $index. */
839: SPL_METHOD(Array, offsetGet)
840: {
841:     zval *index;
842:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
843:         return;
844:     }
845:     return spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
846: } /* }}} */
847:
848: /* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $value)
849:    Sets the value at the specified $index to $value. */
850: SPL_METHOD(Array, offsetSet)
851: {
852:     zval *index, *value;
853:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "aa", &index, &value) == FAILURE) {
854:         return;
855:     }
856:     spl_array_write_dimension_ex(1, getThis(), index, value);
857: } /* }}} */
858:
859: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
860:    Unsets the value at the specified $index. */
861: SPL_METHOD(Array, offsetUnset)
862: {
863:     zval *index;
864:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
865:         return;
866:     }
867:     spl_array_unset_dimension_ex(1, getThis(), index);
868: } /* }}} */
869:
870: /* {{{ proto mixed ArrayObject::offsetIsset(mixed $index)
871:    Returns whether the requested $index exists. */
872: SPL_METHOD(Array, offsetIsset)
873: {
874:     zval *index;
875:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
876:         return;
877:     }
878:     return spl_array_has_dimension_ex(0, getThis(), index, 2);
879: } /* }}} */
880:
881: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
882:    Returns the value at the specified $index. */
883: SPL_METHOD(Array, offsetGet)
884: {
885:     zval *index;
886:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
887:         return;
888:     }
889:     return spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
890: } /* }}} */
891:
892: /* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $value)
893:    Sets the value at the specified $index to $value. */
894: SPL_METHOD(Array, offsetSet)
895: {
896:     zval *index, *value;
897:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "aa", &index, &value) == FAILURE) {
898:         return;
899:     }
900:     spl_array_write_dimension_ex(1, getThis(), index, value);
901: } /* }}} */
902:
903: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
904:    Unsets the value at the specified $index. */
905: SPL_METHOD(Array, offsetUnset)
906: {
907:     zval *index;
908:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
909:         return;
910:     }
911:     spl_array_unset_dimension_ex(1, getThis(), index);
912: } /* }}} */
913:
914: /* {{{ proto mixed ArrayObject::offsetIsset(mixed $index)
915:    Returns whether the requested $index exists. */
916: SPL_METHOD(Array, offsetIsset)
917: {
918:     zval *index;
919:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
920:         return;
921:     }
922:     return spl_array_has_dimension_ex(0, getThis(), index, 2);
923: } /* }}} */
924:
925: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
926:    Returns the value at the specified $index. */
927: SPL_METHOD(Array, offsetGet)
928: {
929:     zval *index;
930:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
931:         return;
932:     }
933:     return spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
934: } /* }}} */
935:
936: /* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $value)
937:    Sets the value at the specified $index to $value. */
938: SPL_METHOD(Array, offsetSet)
939: {
940:     zval *index, *value;
941:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "aa", &index, &value) == FAILURE) {
942:         return;
943:     }
944:     spl_array_write_dimension_ex(1, getThis(), index, value);
945: } /* }}} */
946:
947: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
948:    Unsets the value at the specified $index. */
949: SPL_METHOD(Array, offsetUnset)
950: {
951:     zval *index;
952:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
953:         return;
954:     }
955:     spl_array_unset_dimension_ex(1, getThis(), index);
956: } /* }}} */
957:
958: /* {{{ proto mixed ArrayObject::offsetIsset(mixed $index)
959:    Returns whether the requested $index exists. */
960: SPL_METHOD(Array, offsetIsset)
961: {
962:     zval *index;
963:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
964:         return;
965:     }
966:     return spl_array_has_dimension_ex(0, getThis(), index, 2);
967: } /* }}} */
968:
969: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
970:    Returns the value at the specified $index. */
971: SPL_METHOD(Array, offsetGet)
972: {
973:     zval *index;
974:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
975:         return;
976:     }
977:     return spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
978: } /* }}} */
979:
980: /* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $value)
981:    Sets the value at the specified $index to $value. */
982: SPL_METHOD(Array, offsetSet)
983: {
984:     zval *index, *value;
985:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "aa", &index, &value) == FAILURE) {
986:         return;
987:     }
988:     spl_array_write_dimension_ex(1, getThis(), index, value);
989: } /* }}} */
990:
991: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
992:    Unsets the value at the specified $index. */
993: SPL_METHOD(Array, offsetUnset)
994: {
995:     zval *index;
996:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
997:         return;
998:     }
999:     spl_array_unset_dimension_ex(1, getThis(), index);
1000: } /* }}} */
1001:
1002: /* {{{ proto mixed ArrayObject::offsetIsset(mixed $index)
1003:    Returns whether the requested $index exists. */
1004: SPL_METHOD(Array, offsetIsset)
1005: {
1006:     zval *index;
1007:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1008:         return;
1009:     }
1010:     return spl_array_has_dimension_ex(0, getThis(), index, 2);
1011: } /* }}} */
1012:
1013: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
1014:    Returns the value at the specified $index. */
1015: SPL_METHOD(Array, offsetGet)
1016: {
1017:     zval *index;
1018:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1019:         return;
1020:     }
1021:     return spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
1022: } /* }}} */
1023:
1024: /* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $value)
1025:    Sets the value at the specified $index to $value. */
1026: SPL_METHOD(Array, offsetSet)
1027: {
1028:     zval *index, *value;
1029:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "aa", &index, &value) == FAILURE) {
1030:         return;
1031:     }
1032:     spl_array_write_dimension_ex(1, getThis(), index, value);
1033: } /* }}} */
1034:
1035: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
1036:    Unsets the value at the specified $index. */
1037: SPL_METHOD(Array, offsetUnset)
1038: {
1039:     zval *index;
1040:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1041:         return;
1042:     }
1043:     spl_array_unset_dimension_ex(1, getThis(), index);
1044: } /* }}} */
1045:
1046: /* {{{ proto mixed ArrayObject::offsetIsset(mixed $index)
1047:    Returns whether the requested $index exists. */
1048: SPL_METHOD(Array, offsetIsset)
1049: {
1050:     zval *index;
1051:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1052:         return;
1053:     }
1054:     return spl_array_has_dimension_ex(0, getThis(), index, 2);
1055: } /* }}} */
1056:
1057: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
1058:    Returns the value at the specified $index. */
1059: SPL_METHOD(Array, offsetGet)
1060: {
1061:     zval *index;
1062:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1063:         return;
1064:     }
1065:     return spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
1066: } /* }}} */
1067:
1068: /* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $value)
1069:    Sets the value at the specified $index to $value. */
1070: SPL_METHOD(Array, offsetSet)
1071: {
1072:     zval *index, *value;
1073:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "aa", &index, &value) == FAILURE) {
1074:         return;
1075:     }
1076:     spl_array_write_dimension_ex(1, getThis(), index, value);
1077: } /* }}} */
1078:
1079: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
1080:    Unsets the value at the specified $index. */
1081: SPL_METHOD(Array, offsetUnset)
1082: {
1083:     zval *index;
1084:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1085:         return;
1086:     }
1087:     spl_array_unset_dimension_ex(1, getThis(), index);
1088: } /* }}} */
1089:
1090: /* {{{ proto mixed ArrayObject::offsetIsset(mixed $index)
1091:    Returns whether the requested $index exists. */
1092: SPL_METHOD(Array, offsetIsset)
1093: {
1094:     zval *index;
1095:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1096:         return;
1097:     }
1098:     return spl_array_has_dimension_ex(0, getThis(), index, 2);
1099: } /* }}} */
1100:
1101: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
1102:    Returns the value at the specified $index. */
1103: SPL_METHOD(Array, offsetGet)
1104: {
1105:     zval *index;
1106:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1107:         return;
1108:     }
1109:     return spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
1110: } /* }}} */
1111:
1112: /* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $value)
1113:    Sets the value at the specified $index to $value. */
1114: SPL_METHOD(Array, offsetSet)
1115: {
1116:     zval *index, *value;
1117:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "aa", &index, &value) == FAILURE) {
1118:         return;
1119:     }
1120:     spl_array_write_dimension_ex(1, getThis(), index, value);
1121: } /* }}} */
1122:
1123: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
1124:    Unsets the value at the specified $index. */
1125: SPL_METHOD(Array, offsetUnset)
1126: {
1127:     zval *index;
1128:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1129:         return;
1130:     }
1131:     spl_array_unset_dimension_ex(1, getThis(), index);
1132: } /* }}} */
1133:
1134: /* {{{ proto mixed ArrayObject::offsetIsset(mixed $index)
1135:    Returns whether the requested $index exists. */
1136: SPL_METHOD(Array, offsetIsset)
1137: {
1138:     zval *index;
1139:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1140:         return;
1141:     }
1142:     return spl_array_has_dimension_ex(0, getThis(), index, 2);
1143: } /* }}} */
1144:
1145: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
1146:    Returns the value at the specified $index. */
1147: SPL_METHOD(Array, offsetGet)
1148: {
1149:     zval *index;
1150:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1151:         return;
1152:     }
1153:     return spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
1154: } /* }}} */
1155:
1156: /* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $value)
1157:    Sets the value at the specified $index to $value. */
1158: SPL_METHOD(Array, offsetSet)
1159: {
1160:     zval *index, *value;
1161:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "aa", &index, &value) == FAILURE) {
1162:         return;
1163:     }
1164:     spl_array_write_dimension_ex(1, getThis(), index, value);
1165: } /* }}} */
1166:
1167: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
1168:    Unsets the value at the specified $index. */
1169: SPL_METHOD(Array, offsetUnset)
1170: {
1171:     zval *index;
1172:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1173:         return;
1174:     }
1175:     spl_array_unset_dimension_ex(1, getThis(), index);
1176: } /* }}} */
1177:
1178: /* {{{ proto mixed ArrayObject::offsetIsset(mixed $index)
1179:    Returns whether the requested $index exists. */
1180: SPL_METHOD(Array, offsetIsset)
1181: {
1182:     zval *index;
1183:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1184:         return;
1185:     }
1186:     return spl_array_has_dimension_ex(0, getThis(), index, 2);
1187: } /* }}} */
1188:
1189: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
1190:    Returns the value at the specified $index. */
1191: SPL_METHOD(Array, offsetGet)
1192: {
1193:     zval *index;
1194:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1195:         return;
1196:     }
1197:     return spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
1198: } /* }}} */
1199:
1200: /* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $value)
1201:    Sets the value at the specified $index to $value. */
1202: SPL_METHOD(Array, offsetSet)
1203: {
1204:     zval *index, *value;
1205:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "aa", &index, &value) == FAILURE) {
1206:         return;
1207:     }
1208:     spl_array_write_dimension_ex(1, getThis(), index, value);
1209: } /* }}} */
1210:
1211: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
1212:    Unsets the value at the specified $index. */
1213: SPL_METHOD(Array, offsetUnset)
1214: {
1215:     zval *index;
1216:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1217:         return;
1218:     }
1219:     spl_array_unset_dimension_ex(1, getThis(), index);
1220: } /* }}} */
1221:
1222: /* {{{ proto mixed ArrayObject::offsetIsset(mixed $index)
1223:    Returns whether the requested $index exists. */
1224: SPL_METHOD(Array, offsetIsset)
1225: {
1226:     zval *index;
1227:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1228:         return;
1229:     }
1230:     return spl_array_has_dimension_ex(0, getThis(), index, 2);
1231: } /* }}} */
1232:
1233: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
1234:    Returns the value at the specified $index. */
1235: SPL_METHOD(Array, offsetGet)
1236: {
1237:     zval *index;
1238:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1239:         return;
1240:     }
1241:     return spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
1242: } /* }}} */
1243:
1244: /* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $value)
1245:    Sets the value at the specified $index to $value. */
1246: SPL_METHOD(Array, offsetSet)
1247: {
1248:     zval *index, *value;
1249:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "aa", &index, &value) == FAILURE) {
1250:         return;
1251:     }
1252:     spl_array_write_dimension_ex(1, getThis(), index, value);
1253: } /* }}} */
1254:
1255: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
1256:    Unsets the value at the specified $index. */
1257: SPL_METHOD(Array, offsetUnset)
1258: {
1259:     zval *index;
1260:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1261:         return;
1262:     }
1263:     spl_array_unset_dimension_ex(1, getThis(), index);
1264: } /* }}} */
1265:
1266: /* {{{ proto mixed ArrayObject::offsetIsset(mixed $index)
1267:    Returns whether the requested $index exists. */
1268: SPL_METHOD(Array, offsetIsset)
1269: {
1270:     zval *index;
1271:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &index) == FAILURE) {
1272:         return;
1273:     }
1274:     return spl_array_has_dimension_ex(0, getThis(), index, 2);
1275: } /* }}} */
1276:
1277: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
1278:    Returns the value at the specified $index. */
1279: SPL_METHOD(Array, offsetGet)
1280: {

```

```

753: if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "es", &index, &value) == FAILURE) {
754:     return;
755: }
756: spl_array_write_dimension_ex(0, getThis(), index, value);
757: } /* }}} */
758:
759: void spl_array_iterator_append(zval *object, zval *append_value) /* {{{ */
760: {
761:     spl_array_object *intern = Z_SPLARRAY_P(object);
762:     HashTable *ah = spl_array_get_hash_table(intern);
763:
764:     if (!ah) {
765:         php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
766:         return;
767:     }
768:
769:     if (spl_array_is_object(intern) {
770:         zend_throw_error(NULL, "Cannot append properties to objects, use %s::offsetSet() instead", ZSTR_VAL(Z_OBJCE_P(object)->name));
771:         return;
772:     }
773:
774:     spl_array_write_dimension(object, NULL, append_value);
775: } /* }}} */
776:
777: /* {{{ proto void ArrayObject::append(mixed $value)
778:    proto void ArrayIterator::append(mixed $value)
779:    Appends the value (cannot be called for objects). */
780: #define SPL_METHOD(Array, append)
781:
782: zval *value;
783:
784: if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s", &value) == FAILURE) {
785:     return;
786: }
787: spl_array_iterator_append(getThis(), value);
788: } /* }}} */
789:
790: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
791:    proto void ArrayIterator::offsetUnset(mixed $index)
792:    Unsets the value at the specified $index. */
793: #define SPL_METHOD(Array, offsetUnset)
794:
795: zval *index;
796: if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s", &index) == FAILURE) {
797:     return;
798: }
799: spl_array_unset_dimension_ex(0, getThis(), index);
800: } /* }}} */
801:
802: /* {{{ proto array ArrayObject::getArrayCopy()
803:    proto array ArrayIterator::getArrayCopy()
804:    Returns a copy of the contained array */
805: #define SPL_METHOD(Array, getArrayCopy)
806:
807: zval *object = getThis();
808: spl_array_object *intern = Z_SPLARRAY_P(object);
809:
810: RETURN_ARRAY(zend_array_dup(spl_array_get_hash_table(intern)));
811: } /* }}} */
812:
813: static HashTable *spl_array_get_properties(zval *object) /* {{{ */
814: {
815:     spl_array_object *intern = Z_SPLARRAY_P(object);
816:
817:     if ((intern->var_flags & SPL_ARRAY_STD_PROP_LIST) {
818:         if ((intern->std_properties) {
819:             rebuild_object_properties(&intern->std);
820:         }
821:         return intern->std_properties;
822:     }
823:
824:     return spl_array_get_hash_table(intern);
825: } /* }}} */
826:
827: static HashTable * spl_array_get_debug_info(zval *obj, int *is_temp) /* {{{ */
828: {
829:     zval *storage;
830:     zend_string *name;
831:     zend_string *key;
832:     spl_array_object *intern = Z_SPLARRAY_P(obj);
833:
834:     if ((intern->std_properties) {
835:         rebuild_object_properties(&intern->std);
836:     }
837:
838:     if ((intern->var_flags & SPL_ARRAY_IS_SELF) {
839:         *is_temp = 0;
840:         return intern->std_properties;
841:     } else {
842:         HashTable *debug_info;
843:         *is_temp = 1;
844:
845:         debug_info = zend_new_array(zend_hash_num_elements(intern->std_properties) + 1);
846:         zend_hash_copy(debug_info, intern->std_properties, (copy_ctor_func_t) zval_add_ref);
847:
848:         storage = &intern->array;
849:         Z_TRY_ADDREF_P(storage);
850:
851:         base = Z_OBJ_HT_P(obj) == spl_handler_ArrayIterator
852:             ? spl_obj_ArrayIterator : spl_obj_ArrayObject;
853:         name = spl_get_private_prop_name(base, "storage", &storage) - 1;
854:         zend_symtable_update(debug_info, name, storage);
855:         zend_string_release(name);
856:
857:         return debug_info;
858:     }
859: } /* }}} */
860:
861: static HashTable *spl_array_get_gc(zval *obj, zval **gc_data, int *gc_data_count) /* {{{ */
862: {
863:     spl_array_object *intern = Z_SPLARRAY_P(obj);
864:
865:     *gc_data = &intern->array;
866:     *gc_data_count = 1;
867:     return zend_std_get_properties(obj);
868: } /* }}} */
869:
870: static zval *spl_array_read_property(zval *object, zval *member, int type, void **cache_slot, zval **rv) /* {{{ */
871: {
872:     spl_array_object *intern = Z_SPLARRAY_P(object);
873:
874:     if ((intern->var_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
875:         && !std_object_handlers.has_property(object, member, 2, NULL)) {
876:         return spl_array_read_dimension(object, member, type, rv);
877:     }
878:
879:     return std_object_handlers.read_property(object, member, type, cache_slot, rv);
880: } /* }}} */
881:
882: static void spl_array_write_property(zval *object, zval *member, zval *value, void **cache_slot) /* {{{ */
883: {
884:     spl_array_object *intern = Z_SPLARRAY_P(object);
885:
886:     if ((intern->var_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
887:         && !std_object_handlers.has_property(object, member, 2, NULL)) {
888:         spl_array_write_dimension(object, member, value);
889:         return;
890:     }
891:
892:     std_object_handlers.write_property(object, member, value, cache_slot);
893: } /* }}} */
894:
895: static zval *spl_array_get_property_ptr_ptr(zval *object, zval *member, int type, void **cache_slot) /* {{{ */
896: {
897:     spl_array_object *intern = Z_SPLARRAY_P(object);
898:
899:     if ((intern->var_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
900:         && !std_object_handlers.has_property(object, member, 2, NULL)) {
901:         /* If object has offset() or offset() method, then fallback to read_property,
902:          * which will call offsetGet(). */
903:         if ((intern->flags & SPL_OFFSET_GET) {
904:             return NULL;
905:         }
906:         return spl_array_get_dimension_ptr(1, intern, member, type);
907:     }
908:     return std_object_handlers.get_property_ptr_ptr(object, member, type, cache_slot);
909: } /* }}} */
910:
911: static int spl_array_has_property(zval *object, zval *member, int has_set_exists, void **cache_slot) /* {{{ */
912: {
913:     spl_array_object *intern = Z_SPLARRAY_P(object);
914:
915:     if ((intern->var_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
916:         && !std_object_handlers.has_property(object, member, 2, NULL)) {
917:         return spl_array_has_dimension(object, member, has_set_exists);
918:     }
919:     return std_object_handlers.has_property(object, member, has_set_exists, cache_slot);
920: } /* }}} */
921:
922: static void spl_array_unset_property(zval *object, zval *member, void **cache_slot) /* {{{ */
923: {
924:     spl_array_object *intern = Z_SPLARRAY_P(object);
925:
926:     if ((intern->var_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
927:         && !std_object_handlers.has_property(object, member, 2, NULL)) {
928:         spl_array_unset_dimension(object, member);
929:         return;
930:     }
931:     std_object_handlers.unset_property(object, member, cache_slot);
932: } /* }}} */
933:
934: static int spl_array_compare_objects(zval *o1, zval *o2) /* {{{ */
935: {
936:     HashTable *ht1,
937:               *ht2;
938:     spl_array_object *intern1,
939:                     *intern2;
940:     int result = 0;
941:
942:     intern1 = Z_SPLARRAY_P(o1);
943:     intern2 = Z_SPLARRAY_P(o2);
944:     ht1 = spl_array_get_hash_table(intern1);
945:     ht2 = spl_array_get_hash_table(intern2);
946:
947:     result = zend_compare_symbol_tables(ht1, ht2);
948:     /* If we just compared std_properties, don't do it again */
949:     if (result == 0 &&
950:         (!ht1 == intern1->std_properties && ht2 == intern2->std_properties)) {
951:         result = std_object_handlers.compare_objects(o1, o2);
952:     }
953:     return result;
954: } /* }}} */
955:
956: static int spl_array_skip_protected(spl_array_object *intern, HashTable *ah) /* {{{ */
957: {
958:     zend_string *attr_key;
959:     zend_ulong num_key;
960:     zval *data;
961:
962:     if (spl_array_is_object(intern)) {
963:         uint32_t *pos_ptr = spl_array_get_pos_ptr(ah, intern);
964:
965:         do {
966:             if (zend_hash_get_current_key_ex(ah, attr_key, num_key, pos_ptr) == HASH_KEY_IS_STRING) {
967:                 data = zend_hash_get_current_data_ex(ah, pos_ptr);
968:                 if (data && Z_TYPE_P(data) == IS_INDIRECT) &&
969:                     Z_TYPE_P(*data) == Z_INDIRECT_P(data) == IS_UNDEF) {
970:                     /* skip */
971:                 } else if (!ZSTR_LEN(string_key) || ZSTR_VAL(string_key)[0]) {
972:                     return SUCCESS;
973:                 }
974:             } else {
975:                 return SUCCESS;
976:             }
977:             if (zend_hash_has_more_elements_ex(ah, pos_ptr) != SUCCESS) {
978:                 return FAILURE;
979:             }
980:             zend_hash_move_forward_ex(ah, pos_ptr);
981:         } while (1);
982:         return FAILURE;
983:     } /* }}} */
984:
985:     static int spl_array_next_ex(spl_array_object *intern, HashTable *ah) /* {{{ */
986:     {
987:         uint32_t *pos_ptr = spl_array_get_pos_ptr(ah, intern);
988:
989:         zend_hash_move_forward_ex(ah, pos_ptr);
990:         if (spl_array_is_object(intern)) {
991:             return spl_array_skip_protected(intern, ah);
992:         } else {
993:             return zend_hash_has_more_elements_ex(ah, pos_ptr);
994:         }
995:     } /* }}} */
996:
997:     static int spl_array_next(spl_array_object *intern) /* {{{ */
998:     {
999:         HashTable *ah = spl_array_get_hash_table(intern);
1000:
1001:         return spl_array_next_ex(intern, ah);
1002:     } /* }}} */
1003:
1004:     static void spl_array_iterator_ctor(zend_object_iterator *iter) /* {{{ */
1005:     {
1006:         zend_user_it_initialize_current(iter);
1007:         zval_ptr_dtor(&iter->data);
1008:         iter->data = spl_array_get_hash_table(intern);
1009:     } /* }}} */
1010:
1011:     static int spl_array_iterator_valid(zend_object_iterator *iter) /* {{{ */
1012:     {
1013:         spl_array_object *object = Z_SPLARRAY_P(iter->data);
1014:         HashTable *ah = spl_array_get_hash_table(object);
1015:
1016:         if (object->var_flags & SPL_ARRAY_OVERLOADED_VALID) {
1017:             return zend_user_it_valid(iter);
1018:         } else {
1019:             if (spl_array_object_verify_pos_ex(object, ah, "ArrayIterator::valid()") == FAILURE) {
1020:                 return FAILURE;
1021:             }
1022:             return zend_hash_has_more_elements_ex(ah, spl_array_get_pos_ptr(ah, object));
1023:         }
1024:     } /* }}} */
1025:
1026:     static zval *spl_array_iterator_get_current_data(zend_object_iterator *iter) /* {{{ */
1027:     {
1028:         spl_array_object *object = Z_SPLARRAY_P(iter->data);
1029:         HashTable *ah = spl_array_get_hash_table(object);
1030:
1031:         if (object->var_flags & SPL_ARRAY_OVERLOADED_CURRENT) {
1032:             return zend_user_it_get_current_data(iter);
1033:         } else {
1034:             zval *data = zend_hash_get_current_data_ex(ah, spl_array_get_pos_ptr(ah, object));
1035:             if (Z_TYPE_P(data) == IS_INDIRECT) {
1036:                 data = Z_INDIRECT_P(data);
1037:             }
1038:             return data;
1039:         }
1040:     } /* }}} */
1041:
1042:     static void spl_array_iterator_get_current_key(zend_object_iterator *iter, zval **key) /* {{{ */
1043:     {
1044:         spl_array_object *object = Z_SPLARRAY_P(iter->data);
1045:         HashTable *ah = spl_array_get_hash_table(object);
1046:
1047:         if (object->var_flags & SPL_ARRAY_OVERLOADED_KEY) {
1048:             zend_user_it_get_current_key(iter, key);
1049:         } else {
1050:             if (spl_array_object_verify_pos_ex(object, ah, "ArrayIterator::current()") == FAILURE) {
1051:                 return NULL;
1052:             }
1053:             zend_hash_get_current_key_zval_ex(ah, key, spl_array_get_pos_ptr(ah, object));
1054:         }
1055:     } /* }}} */
1056:
1057:     static void spl_array_iterator_move_forward(zend_object_iterator *iter) /* {{{ */
1058:     {
1059:         spl_array_object *object = Z_SPLARRAY_P(iter->data);
1060:         HashTable *ah = spl_array_get_hash_table(object);
1061:
1062:         if (object->var_flags & SPL_ARRAY_OVERLOADED_NEXT) {
1063:             zend_user_it_move_forward(iter);
1064:         } else {
1065:             zend_user_it_invalidate_current(iter);
1066:             if (!ah) {
1067:                 php_error_docref(NULL, E_NOTICE, "ArrayIterator::current(): Array was modified outside object and is no longer an array");
1068:                 return;
1069:             }
1070:             spl_array_next_ex(object, ah);
1071:         }
1072:     } /* }}} */
1073:
1074:     static void spl_array_iterator_rewind(spl_array_object *intern) /* {{{ */
1075:     {
1076:         HashTable *ah = spl_array_get_hash_table(intern);
1077:
1078:         if (!ah) {
1079:             php_error_docref(NULL, E_NOTICE, "ArrayIterator::rewind(): Array was modified outside object and is no longer an array");
1080:             return;
1081:         }
1082:         if (intern->ah_iter == (uint32_t)-1) {
1083:             spl_array_get_pos_ptr(ah, intern);
1084:         } else {
1085:             zend_hash_internal_pointer_reset_ex(ah, spl_array_get_pos_ptr(ah, intern));
1086:             spl_array_skip_protected(intern, ah);
1087:         }
1088:     } /* }}} */
1089:
1090:     static void spl_array_iterator_rewind(zend_object_iterator *iter) /* {{{ */
1091:     {
1092:         spl_array_object *object = Z_SPLARRAY_P(iter->data);
1093:
1094:         if (object->var_flags & SPL_ARRAY_OVERLOADED_REWIND) {
1095:             zend_user_it_rewind(iter);
1096:         } else {
1097:             zend_user_it_invalidate_current(iter);
1098:             spl_array_rewind(object);
1099:         }
1100:     } /* }}} */
1101:
1102:     static void spl_array_iterator_set_array(zval *object, spl_array_object *intern, zval *array, zend_ulong ar_flags, int just_array) /* {{{ */
1103:     {
1104:         if (Z_TYPE_P(array) != IS_ARRAY) {
1105:             zend_throw_exception(spl_ce_invalid_argument_exception, "Passed variable is not an array or object", 0);
1106:             return;
1107:         }
1108:
1109:         if (Z_TYPE_P(array) == IS_ARRAY) {
1110:             zval_ptr_dtor(&intern->array);
1111:             Z_ADDREF_P(array);
1112:             zend_copy(&intern->array, array);
1113:         } else {
1114:             /* ??? TODO: try to avoid array duplication */
1115:             ZVAL_ARR(&intern->array, zend_array_dup(Z_ARR_P(array)));
1116:         }
1117:     }

```

```

1: if (Z_OBJ_HT_P(array) == spl_handler_ArrayObject { 1_Z_OBJ_HT_P(array) == spl_handler_ArrayIterator) {
2:     zval_ptr_dtor(&intern->array);
3:     if (!zval_array(array)) {
4:         spl_array_object *other = 2_SPLARRAY_P(array);
5:         ar_flags = other->ar_flags & "SPL_ARRAY_INT_MASK";
6:     }
7:     if (Z_OBJ_P(object) == 2_OBJ_P(array)) {
8:         ar_flags = SPL_ARRAY_IS_SELF;
9:         ZVAL_UNDEF(&intern->array);
10:    } else {
11:        ar_flags = SPL_ARRAY_USE_OTHER;
12:        ZVAL_COPY(&intern->array, array);
13:    }
14: } else {
15:     zend_object_get_properties_t_handler = 2_OBJ_HANDLER_P(array, get_properties);
16:     if (handler != std_object_handlers.get_properties()) {
17:         zend_throw_exception_ex(spl_ce_invalidArgumentException, 0,
18:             "Overloaded object of type %s is not compatible with %s",
19:             ESTR_VAL(Z_OBJCE_P(array)->name), ESTR_VAL(intern->std.ce->name));
20:         return;
21:     }
22: }
23:
24: intern->ar_flags = "SPL_ARRAY_IS_SELF & SPL_ARRAY_USE_OTHER;
25: intern->ar_flags |= ar_flags;
26: intern->znt_iter = (uint32_t)-1;
27: }
28:
29: /* }}} */
30:
31: /* iterator handler table */
32: static const zend_object_iterator_funcs spl_array_it_funcs = {
33:     spl_array_it_dtor,
34:     spl_array_it_valid,
35:     spl_array_it_get_current_data,
36:     spl_array_it_get_next_key,
37:     spl_array_it_move_forward,
38:     spl_array_it_rewind,
39:     NULL
40: };
41:
42: zend_object_iterator *spl_array_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
43: {
44:     zend_user_iterator *iterator;
45:     zend_array_object *array_object = 2_SPLARRAY_P(object);
46:
47:     if (by_ref & (array_object->ar_flags & SPL_ARRAY_OVERLOADED_CURRENT)) {
48:         zend_throw_exception(spl_ce_runtime_exception, "No iterator cannot be used with foreach by reference", 0);
49:         return NULL;
50:     }
51:
52:     iterator = emalloc(sizeof(zend_user_iterator));
53:
54:     zend_iterator_init(iterator->it);
55:
56:     ZVAL_COPY(&iterator->it.data, object);
57:     iterator->it.funcs = spl_array_it_funcs;
58:     iterator->ce = ce;
59:     ZVAL_UNDEF(iterator->value);
60:
61:     return iterator->it;
62: }
63:
64: /* }}} */
65:
66: /* {{{ proto void ArrayObject::__construct([array|object ar = array() [, int flags = 0 [, string iterator_class]])
67:    Constructs a new array iterator from an array or object. */
68: SPL_METHOD(Array, __construct)
69: {
70:     zval *object = &this();
71:     spl_array_object *intern;
72:     zval *array;
73:     zend_long ar_flags = 0;
74:     zend_class_entry *ce_get_iterator = spl_ce_iterator;
75:
76:     if (ZEND_NUM_ARGS() == 0) {
77:         return; /* nothing to do */
78:     }
79:
80:     if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "e|l", &array, &ar_flags, &ce_get_iterator) == FAILURE) {
81:         return;
82:     }
83:
84:     intern = 2_SPLARRAY_P(object);
85:
86:     if (ZEND_NUM_ARGS() > 2) {
87:         intern->ce_get_iterator = ce_get_iterator;
88:     }
89:
90:     ar_flags |= "SPL_ARRAY_INT_MASK;
91:     spl_array_set_array(object, intern, array, ar_flags, ZEND_NUM_ARGS() == 1);
92: }
93:
94: /* }}} */
95:
96: /* {{{ proto void ArrayIterator::__construct([array|object ar = array() [, int flags = 0])
97:    Constructs a new array iterator from an array or object. */
98: SPL_METHOD(ArrayIterator, __construct)
99: {
100:     zval *object = &this();
101:     spl_array_object *intern;
102:     zval *array;
103:     zend_long ar_flags = 0;
104:
105:     if (ZEND_NUM_ARGS() == 0) {
106:         return; /* nothing to do */
107:     }
108:
109:     if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "e|l", &array, &ar_flags) == FAILURE) {
110:         return;
111:     }
112:
113:     intern = 2_SPLARRAY_P(object);
114:
115:     ar_flags |= "SPL_ARRAY_INT_MASK;
116:     spl_array_set_array(object, intern, array, ar_flags, ZEND_NUM_ARGS() == 1);
117: }
118:
119: /* }}} */
120:
121: /* {{{ proto void ArrayObject::setIteratorClass(string iterator_class)
122:    Set the class used in getIterator. */
123: SPL_METHOD(Array, setIteratorClass)
124: {
125:     zval *object = &this();
126:     spl_array_object *intern = 2_SPLARRAY_P(object);
127:     zval *array;
128:     zend_class_entry *ce_get_iterator = spl_ce_iterator;
129:
130:     ZEND_PARSE_PARAMETERS_START(1, 1)
131:     Z_PARAM_CLASS(ce_get_iterator)
132:     ZEND_PARSE_PARAMETERS_END();
133:
134:     intern->ce_get_iterator = ce_get_iterator;
135: }
136:
137: /* }}} */
138:
139: /* {{{ proto string ArrayObject::getIteratorClass()
140:    Get the class used in getIterator. */
141: SPL_METHOD(Array, getIteratorClass)
142: {
143:     zval *object = &this();
144:     spl_array_object *intern = 2_SPLARRAY_P(object);
145:
146:     if (zend_parse_parameters_none() == FAILURE) {
147:         return;
148:     }
149:
150:     zend_string_offset(intern->ce_get_iterator->name);
151:     RETURN_STR(intern->ce_get_iterator->name);
152: }
153:
154: /* }}} */
155:
156: /* {{{ proto int ArrayObject::getFlags()
157:    Get flags */
158: SPL_METHOD(Array, getFlags)
159: {
160:     zval *object = &this();
161:     spl_array_object *intern = 2_SPLARRAY_P(object);
162:
163:     if (zend_parse_parameters_none() == FAILURE) {
164:         return;
165:     }
166:
167:     RETURN_LONG(intern->ar_flags & "SPL_ARRAY_INT_MASK;
168: }
169:
170: /* }}} */
171:
172: /* {{{ proto void ArrayObject::setFlags(int flags)
173:    Set flags */
174: SPL_METHOD(Array, setFlags)
175: {
176:     zval *object = &this();
177:     spl_array_object *intern = 2_SPLARRAY_P(object);
178:     zend_long ar_flags = 0;
179:
180:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &ar_flags) == FAILURE) {
181:         return;
182:     }
183:
184:     intern->ar_flags = (intern->ar_flags & SPL_ARRAY_INT_MASK) | (ar_flags & "SPL_ARRAY_INT_MASK;
185: }
186:
187: /* }}} */
188:
189: /* {{{ proto ArrayObject ArrayObject::exchangeArray(ArrayObject ar = array())
190:    Replace the referenced array or object with a new one and return the old one (right now copy - to be changed)
191:    Array, exchangeArray)
192: }

```

```

3117:     sval *object = getThis(), *array;
3118:     spl_array_object *intern = _S_PLARRAY_P(object);
3119: }
3120:
3121: zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "a", &array) == FAILURE {
3122:     return;
3123: }
3124:
3125: if (intern->mapIteratorCount > 0) {
3126:     zend_error(E_NOTICE, "Modification of ArrayObject during sorting is prohibited");
3127: }
3128:
3129: RETVAL_ARR(zend_array_dup(spl_array_get_hash_table(intern)));
3130: spl_array_set_array(object, intern, array, 0, 1);
3131: }
3132: /* }}} */
3133:
3134: /* {{{ proto ArrayIterator ArrayObject::getIterator()
3135:  * Create a new iterator from a ArrayObject instance */
3136: SPL_METHOD(Array, getIterator)
3137: {
3138:     sval *object = getThis();
3139:     spl_array_object *intern = _S_PLARRAY_P(object);
3140:     HashTable *ah = spl_array_get_hash_table(intern);
3141:
3142:     if (zend_parse_parameters_none() == FAILURE) {
3143:         return;
3144:     }
3145:
3146:     if (finfo) {
3147:         php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
3148:         return;
3149:     }
3150:
3151:     ZVAL_OBJ(&return_value, spl_array_object_new_as(intern->obj_get_iterator(), object, 0));
3152: }
3153: /* }}} */
3154:
3155: /* {{{ proto void ArrayIterator::rewind()
3156:  * Rewind array back to the start */
3157: SPL_METHOD(Array, rewind)
3158: {
3159:     sval *object = getThis();
3160:     spl_array_object *intern = _S_PLARRAY_P(object);
3161:
3162:     if (zend_parse_parameters_none() == FAILURE) {
3163:         return;
3164:     }
3165:
3166:     spl_array_rewind(intern);
3167: }
3168: /* }}} */
3169:
3170: /* {{{ proto void ArrayIterator::seek(int $position)
3171:  * Seek to position. */
3172: SPL_METHOD(Array, seek)
3173: {
3174:     zend_long opos, position;
3175:     sval *object = getThis();
3176:     spl_array_object *intern = _S_PLARRAY_P(object);
3177:     HashTable *ah = spl_array_get_hash_table(intern);
3178:     int result;
3179:
3180:     if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "i", &position) == FAILURE) {
3181:         return;
3182:     }
3183:
3184:     if (finfo) {
3185:         php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
3186:         return;
3187:     }
3188:
3189:     opos = position;
3190:
3191:     if (position == 0) { /* negative values are not supported */
3192:         spl_array_rewind(intern);
3193:         result = SUCCESS;
3194:
3195:         while (position-- > 0 && (result = spl_array_next(intern)) == SUCCESS;
3196:
3197:     if (result == SUCCESS && zend_hash_has_more_elements_ex(ah, spl_array_get_pos_ptr(ah, intern)) == SUCCESS) {
3198:         return; /* ok */
3199:     }
3200: }
3201:
3202: zend_throw_exception_ex(spl_ce_OutOfBoundsException, 0, "Seek position " ZEND_LONG_FMT " is out of range", opos);
3203: /* }}} */
3204:
3205: static int spl_array_object_count_elements_helper(spl_array_object *intern, zend_long *count) /* {{{ */
3206: {
3207:     HashTable *ah = spl_array_get_hash_table(intern);
3208:     HashPosition pos, pos_ptr;
3209:
3210:     if (finfo) {
3211:         php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
3212:         *count = 0;
3213:         return FAILURE;
3214:     }
3215:
3216:     if (spl_array_is_object(intern)) {
3217:         /* we need to store the 'pos' since we'll modify it in the functions
3218:          * we're going to call and which do not support 'pos' as parameter. */
3219:         pos_ptr = spl_array_get_pos_ptr(intern);
3220:         pos = *pos_ptr;
3221:         *count = 0;
3222:         spl_array_rewind(intern);
3223:         while (pos_ptr != HT_INVALID_IDX && spl_array_next(intern) == SUCCESS) {
3224:             (*count)++;
3225:         }
3226:         *pos_ptr = pos;
3227:         return SUCCESS;
3228:     } else {
3229:         *count = zend_hash_num_elements(ah);
3230:         return SUCCESS;
3231:     }
3232: }
3233: /* }}} */
3234:
3235: int spl_array_object_count_elements(sval *object, zend_long *count) /* {{{ */
3236: {
3237:     spl_array_object *intern = _S_PLARRAY_P(object);
3238:
3239:     if (intern->fpPtr_count) {
3240:         sval rv;
3241:         zend_call_method_with_0_params(object, intern->std_obj, intern->fpPtr_count, "count", &rv);
3242:         if (IS_TYPE_IV) {
3243:             *count = sval_get_long(rv);
3244:             zend_ptr_error(rv);
3245:             return SUCCESS;
3246:         }
3247:         *count = 0;
3248:         return FAILURE;
3249:     }
3250:     return spl_array_object_count_elements_helper(intern, count);
3251: }
3252: /* }}} */
3253:
3254: /* {{{ proto int ArrayObject::count()
3255:  * proto int ArrayIterator::count()
3256:  * Return the number of elements in the Iterator. */
3257: SPL_METHOD(Array, count)
3258: {
3259:     zend_long count;
3260:     spl_array_object *intern = _S_PLARRAY_P(getThis());
3261:
3262:     if (zend_parse_parameters_none() == FAILURE) {
3263:         return;
3264:     }
3265:
3266:     spl_array_object_count_elements_helper(intern, &count);
3267:
3268:     RETVAL_LONG(count);
3269: }
3270: /* }}} */
3271:
3272: static void spl_array_method_INTERNAL_FUNCTION_PARAMETERS, char *fname, int fname_len, int use_arg) /* {{{ */
3273: {
3274:     spl_array_object *intern = _S_PLARRAY_P(getThis());
3275:     HashTable *ah = spl_array_get_hash_table(intern);
3276:     zend_function_name, param[2], *arg = NULL;
3277:
3278:     ZVAL_STRINGW(function_name, fname, fname_len);
3279:
3280:     ZVAL_NULL(&param[0]);
3281:     ZVAL_STR(&param[0], &ah);
3282:     GC_ADDREF(ah);
3283:
3284:     if (use_arg) {
3285:         ZVAL_COPY_VALUE(&param[1], arg);
3286:     }
3287:
3288:     intern->mapIteratorCount++;
3289:     call_user_function_ex(EG(function_table), NULL, &function_name, return_value, 1, param, 1, NULL);
3290:     if (use_arg == SPL_ARRAY_METHOD_MAY_BEER_ARG) {
3291:         if (zend_parse_parameters_ex(EG(function_table), &intern->P_PARAMS_COUNT, "i", &arg) == FAILURE) {
3292:             zend_throw_exception(spl_ce_BadMethodCallException, "Function expects exactly one argument", 0);
3293:             goto exit;
3294:         }
3295:         if (arg) {
3296:             ZVAL_COPY_VALUE(&param[1], arg);
3297:         }
3298:         call_user_function_ex(EG(function_table), NULL, &function_name, return_value, arg ? 2 : 1, param, 1, NULL);
3299:         intern->mapIteratorCount++;
3300:     } else {
3301:         if (ZEND_NUM_ARGS() != 1) {
3302:             zend_parse_parameters_ex(EG(function_table), &intern->P_PARAMS_COUNT, "i", &arg) == FAILURE) {
3303:                 zend_throw_exception(spl_ce_BadMethodCallException, "Function expects exactly one argument", 0);
3304:                 goto exit;
3305:             }
3306:             ZVAL_COPY_VALUE(&param[1], arg);
3307:             call_user_function_ex(EG(function_table), NULL, &function_name, return_value, 2, param, 1, NULL);
3308:             intern->mapIteratorCount++;
3309:         }
3310:     }
3311: }
3312:

```

```

1500: struct {
1501:     HashTable *new_ht = 2_ARRAYVAL_P(2_REFVAL(params[0]));
1502:     if (aht != new_ht) {
1503:         spl_array_wipeout_hash_table(intern, new_ht);
1504:     } else {
1505:         GC_DEREF(aht);
1506:     }
1507:     cfree(2_REF(params[0]));
1508:     zend_string_free(2_STR(function_name));
1509: }
1510: /* }}} */
1511: #define SPL_ARRAY_METHOD(name, fname, usa_arg) \
1512:     \
1513:     \
1514:     \
1515:     \
1516:     \
1517:     \
1518:     \
1519:     \
1520:     \
1521:     \
1522:     \
1523:     \
1524:     \
1525:     \
1526:     \
1527:     \
1528:     \
1529:     \
1530:     \
1531:     \
1532:     \
1533:     \
1534:     \
1535:     \
1536:     \
1537:     \
1538:     \
1539:     \
1540:     \
1541:     \
1542:     \
1543:     \
1544:     \
1545:     \
1546:     \
1547:     \
1548:     \
1549:     \
1550:     \
1551:     \
1552:     \
1553:     \
1554:     \
1555:     \
1556:     \
1557:     \
1558:     \
1559:     \
1560:     \
1561:     \
1562:     \
1563:     \
1564:     \
1565:     \
1566:     \
1567:     \
1568:     \
1569:     \
1570:     \
1571:     \
1572:     \
1573:     \
1574:     \
1575:     \
1576:     \
1577:     \
1578:     \
1579:     \
1580:     \
1581:     \
1582:     \
1583:     \
1584:     \
1585:     \
1586:     \
1587:     \
1588:     \
1589:     \
1590:     \
1591:     \
1592:     \
1593:     \
1594:     \
1595:     \
1596:     \
1597:     \
1598:     \
1599:     \
1600:     \
1601:     \
1602:     \
1603:     \
1604:     \
1605:     \
1606:     \
1607:     \
1608:     \
1609:     \
1610:     \
1611:     \
1612:     \
1613:     \
1614:     \
1615:     \
1616:     \
1617:     \
1618:     \
1619:     \
1620:     \
1621:     \
1622:     \
1623:     \
1624:     \
1625:     \
1626:     \
1627:     \
1628:     \
1629:     \
1630:     \
1631:     \
1632:     \
1633:     \
1634:     \
1635:     \
1636:     \
1637:     \
1638:     \
1639:     \
1640:     \
1641:     \
1642:     \
1643:     \
1644:     \
1645:     \
1646:     \
1647:     \
1648:     \
1649:     \
1650:     \
1651:     \
1652:     \
1653:     \
1654:     \
1655:     \
1656:     \
1657:     \
1658:     \
1659:     \
1660:     \
1661:     \
1662:     \
1663:     \
1664:     \
1665:     \
1666:     \
1667:     \
1668:     \
1669:     \
1670:     \
1671:     \
1672:     \
1673:     \
1674:     \
1675:     \
1676:     \
1677:     \
1678:     \
1679:     \
1680:     \
1681:     \
1682:     \
1683:     \
1684:     \
1685:     \
1686:     \
1687:     \
1688:     \
1689:     \
1690:     \
1691:     \
1692:     \
1693:     \
1694:     \
1695:     \
1696:     \
1697:     \
1698:     \
1699:     \
1700:     \
1701:     \
1702:     \
1703:     \
1704:     \
1705:     \
1706:     \
1707:     \
1708:     \
1709:     \
1710:     \
1711:     \
1712:     \
1713:     \
1714:     \
1715:     \
1716:     \
1717:     \
1718:     \
1719:     \
1720:     \
1721:     \
1722:     \
1723:     \
1724:     \
1725:     \
1726:     \
1727:     \
1728:     \
1729:     \
1730:     \
1731:     \
1732:     \
1733:     \
1734:     \
1735:     \
1736:     \
1737:     \
1738:     \
1739:     \
1740:     \
1741:     \
1742:     \
1743:     \
1744:     \
1745:     \
1746:     \
1747:     \
1748:     \
1749:     \
1750:     \
1751:     \
1752:     \
1753:     \
1754:     \
1755:     \
1756:     \
1757:     \
1758:     \
1759:     \
1760:     \
1761:     \
1762:     \
1763:     \
1764:     \
1765:     \
1766:     \
1767:     \
1768:     \
1769:     \
1770:     \
1771:     \
1772:     \
1773:     \
1774:     \
1775:     \
1776:     \
1777:     \
1778:     \
1779:     \
1780:     \
1781:     \
1782:     \
1783:     \
1784:     \
1785:     \
1786:     \
1787:     \
1788:     \
1789:     \
1790:     \
1791:     \
1792:     \
1793:     \
1794:     \
1795:     \
1796:     \
1797:     \
1798:     \
1799:     \
1800:     \
1801:     \
1802:     \
1803:     \
1804:     \
1805:     \
1806:     \
1807:     \
1808:     \
1809:     \
1810:     \
1811:     \
1812:     \
1813:     \
1814:     \
1815:     \
1816:     \
1817:     \
1818:     \
1819:     \
1820:     \
1821:     \
1822:     \
1823:     \
1824:     \
1825:     \
1826:     \
1827:     \
1828:     \
1829:     \
1830:     \
1831:     \
1832:     \
1833:     \
1834:     \
1835:     \
1836:     \
1837:     \
1838:     \
1839:     \
1840:     \
1841:     \
1842:     \
1843:     \
1844:     \
1845:     \
1846:     \
1847:     \
1848:     \
1849:     \
1850:     \
1851:     \
1852:     \
1853:     \
1854:     \
1855:     \
1856:     \
1857:     \
1858:     \
1859:     \
1860:     \
1861:     \
1862:     \
1863:     \
1864:     \
1865:     \
1866:     \
1867:     \
1868:     \
1869:     \
1870:     \
1871:     \
1872:     \
1873:     \
1874:     \
1875:     \
1876:     \
1877:     \
1878:     \
1879:     \
1880:     \
1881:     \
1882:     \
1883:     \
1884:     \
1885:     \
1886:     \
1887:     \
1888:     \
1889:     \
1890:     \
1891:     \
1892:     \
1893:     \
1894:     \
1895:     \
1896:     \
1897:     \
1898:     \
1899:     \
1900:     \
1901:     \
1902:     \
1903:     \
1904:     \
1905:     \
1906:     \
1907:     \
1908:     \
1909:     \
1910:     \
1911:     \
1912:     \
1913:     \
1914:     \
1915:     \
1916:     \
1917:     \
1918:     \
1919:     \
1920:     \
1921:     \
1922:     \
1923:     \
1924:     \
1925:     \
1926:     \
1927:     \
1928:     \
1929:     \
1930:     \
1931:     \
1932:     \
1933:     \
1934:     \
1935:     \
1936:     \
1937:     \
1938:     \
1939:     \
1940:     \
1941:     \
1942:     \
1943:     \
1944:     \
1945:     \
1946:     \
1947:     \
1948:     \
1949:     \
1950:     \
1951:     \
1952:     \
1953:     \
1954:     \
1955:     \
1956:     \
1957:     \
1958:     \
1959:     \
1960:     \
1961:     \
1962:     \
1963:     \
1964:     \
1965:     \
1966:     \
1967:     \
1968:     \
1969:     \
1970:     \
1971:     \
1972:     \
1973:     \
1974:     \
1975:     \
1976:     \
1977:     \
1978:     \
1979:     \
1980:     \
1981:     \
1982:     \
1983:     \
1984:     \
1985:     \
1986:     \
1987:     \
1988:     \
1989:     \
1990:     \
1991:     \
1992:     \
1993:     \
1994:     \
1995:     \
1996:     \
1997:     \
1998:     \
1999:     \
2000:     \
2001:     \
2002:
```

```

16891:
16892:
16893:     if (entry == send_ssh_get_current_data_axiath, spi_array_get_pos_ptr(ht, intern)) == NULL) {
16894:         return;
16895:     }
16896:
16897:     if (IS_TYPE_P(entry) == IS_INDIRECT) {
16898:         entry = IS_INDIRECT_P(entry);
16899:     }
16900:
16901:     EVAL_UNDEF(entry);
16902:
16903:     if (IS_TYPE_P(entry) == IS_OBJECT) {
16904:         if ((intern->var_flags & SPL_ARRAY_CHILD_ARRAYS_ONLY) != 0) {
16905:             return;
16906:         }
16907:
16908:         if (instanceof_function(S_OBJECT_P(entry), S_OBJECT_P(getthis()))) {
16909:             EVAL_COPY(return_value, S_OBJ_P(entry));
16910:             S_ADOBEF_P(return_value);
16911:             return;
16912:         }
16913:     }
16914:
16915:     EVAL_LONG(flags, intern->var_flags);
16916:     spi_instantiate_arg_ax2(S_OBJECT_P(getthis()), return_value, entry, flags);
16917: }
16918:
16919: /* }}} */
16920:
16921: /* {{{ proto string ArrayObject::serialize()
16922:  * Serialize the object */
16923:
16924: SPL_METHOD(Array, serialize)
16925: {
16926:     zval *object = getThis();
16927:     spi_array_object *intern = S_SPLARRAY_P(object);
16928:     HashTable *ht = spi_array_get_hash_table(intern);
16929:     zend_members *flags;
16930:     php_serialize_data_t var_hash;
16931:     smart_str buf = {0};
16932:
16933:     if (zend_parse_parameters_none() == FAILURE) {
16934:         return;
16935:     }
16936:
16937:     if (ht) {
16938:         zend_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
16939:         return;
16940:     }
16941:
16942:     PHP_VAR_SERIALIZE_INIT(var_hash);
16943:
16944:     EVAL_LONG(flags, (intern->var_flags & SPL_ARRAY_CLONE_MASK));
16945:
16946:     /* storage */
16947:     smart_str_append(&buf, "a:", 2);
16948:     php_var_serialize(&buf, &flags, &var_hash);
16949:
16950:     if ((intern->var_flags & SPL_ARRAY_IS_SELF) &
16951:         php_var_serialize(&buf, &intern->array, &var_hash)) {
16952:         smart_str_append(&buf, ' ');
16953:     }
16954:
16955:     /* members */
16956:     smart_str_append(&buf, "m:", 2);
16957:     if ((intern->std.properties) &
16958:         rebuild_obj_std_properties(&intern->std)) {
16959:     }
16960:
16961:     EVAL_ARR(&members, intern->std.properties);
16962:     php_var_serialize(&buf, &members, &var_hash); /* finishes the string */
16963:
16964:     /* done */
16965:     PHP_VAR_SERIALIZE_DESTROY(var_hash);
16966:
16967:     if (buf.s) {
16968:         RETURN_NEW_STR(buf.s);
16969:     }
16970:
16971:     RETURN_NULL();
16972: } /* }}} */
16973:
16974: /* {{{ proto void ArrayObject::unserialize(string serialized)
16975:  * unserialize the object
16976:  */
16977:
16978: SPL_METHOD(Array, unserialize)
16979: {
16980:     zval *object = getThis();
16981:     spi_array_object *intern = S_SPLARRAY_P(object);
16982:
16983:     char *buf;
16984:     const buf_len;
16985:     const unsigned char *p;
16986:     php_unserialize_data_t var_hash;
16987:     zend_members *flags;
16988:     zend_long flags;
16989:
16990:     if (zend_parse_parameters(ZEND_NUM_ARGS() & "s", &buf, &buf_len) == FAILURE) {
16991:         return;
16992:     }
16993:
16994:     if (buf_len == 0) {
16995:         return;
16996:     }
16997:
16998:     if (intern->copyCount > 0) {
16999:         zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
17000:         return;
17001:     }
17002:
17003:     /* storage */
17004:     s = p = (const unsigned char *)buf;
17005:     PHP_VAR_UNSERIALIZE_INIT(var_hash);
17006:
17007:     if (*p != 'a' || **p != ':') {
17008:         goto outexcept;
17009:     }
17010:
17011:     *p++;
17012:
17013:     &flags = var_tmp_var(&var_hash);
17014:     if (php_var_unserialize(&flags, &p, s + buf_len, &var_hash) || IS_TYPE_P(&flags) != IS_LONG) {
17015:     }
17016:
17017:     /*
17018:      * "p" is for "p"
17019:      * flags = S_ARRAY_P(&flags);
17020:      * flags needs to be verified and we also need to verify whether the next
17021:      * thing we get is "p". After that we require an "a" or something else
17022:      * where "a" stands for members and anything else should be an array. If
17023:      * neither "a" or "a" follows we have an error. */
17024:
17025:     if (*p != ':') {
17026:         goto outexcept;
17027:     }
17028:
17029:     *p++;
17030:
17031:     if (flags & SPL_ARRAY_IS_SELF) {
17032:         /* If IS_SELF is used, the flags are not followed by an array/object */
17033:         &intern->var_flags = SPL_ARRAY_CLONE_MASK;
17034:         &intern->var_flags |= flags & SPL_ARRAY_CLONE_MASK;
17035:         &var_hash.get_std_for(flags & flags & SPL_ARRAY_CLONE_MASK);
17036:         EVAL_UNDEF(&intern->array);
17037:     } else {
17038:         if (*p != 'a' || **p != 'O' || *p != 'O' || *p != 'C') {
17039:             goto outexcept;
17040:         }
17041:
17042:         array = var_tmp_var(&var_hash);
17043:         if (php_var_unserialize(array, &p, s + buf_len, &var_hash)) {
17044:             if (IS_TYPE_P(array) != IS_ARRAY & IS_TYPE_P(array) != IS_OBJECT) {
17045:                 goto outexcept;
17046:             }
17047:
17048:             &intern->var_flags = SPL_ARRAY_CLONE_MASK;
17049:             &intern->var_flags |= flags & SPL_ARRAY_CLONE_MASK;
17050:
17051:             if (IS_TYPE_P(array) == IS_ARRAY) {
17052:                 &var_hash.get_std_for(&intern->array);
17053:                 EVAL_COPY(&intern->array, array);
17054:             } else {
17055:                 spi_array_set_array(object, &intern, array, 0, 1);
17056:             }
17057:
17058:             if (*p != ':') {
17059:                 goto outexcept;
17060:             }
17061:
17062:             *p++;
17063:
17064:             /* members */
17065:             if (*p != 'm' || **p != ':') {
17066:                 goto outexcept;
17067:             }
17068:
17069:             *p++;
17070:
17071:             &members = var_tmp_var(&var_hash);
17072:             if (php_var_unserialize(&members, &p, s + buf_len, &var_hash) || IS_TYPE_P(&members) != IS_ARRAY) {
17073:                 goto outexcept;
17074:             }
17075:
17076:             /* copy members */
17077:             object_properties_load(&intern->std, &INTERNAL_P(&members));
17078:
17079:             /* done reading Serialized */
17080:             PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
17081:
17082:             outexcept;
17083:
17084:             zend_throw_exception(spl_ce UnexpectedValueException, 0, "Error at offset " ZEND_LONG_FMT

```

```

1881:     return;
1882:
1883: } /* }}} */
1884:
1885: /* {{{ arginfo and function table */
1886: ZEND_BEGIN_ARG_INFO_EX(arginfo_array___construct, 0, 0, 0)
1887:     ZEND_ARG_INFO(0, array)
1888:     ZEND_ARG_INFO(0, ar_flags)
1889:     ZEND_ARG_INFO(0, iterator_class)
1890: ZEND_END_ARG_INFO()
1891:
1892: /* ArrayIterator::__construct and ArrayObject::__construct have different signatures */
1893: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_iterator___construct, 0, 0, 0)
1894:     ZEND_ARG_INFO(0, array)
1895:     ZEND_ARG_INFO(0, ar_flags)
1896: ZEND_END_ARG_INFO()
1897:
1898: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_offsetGet, 0, 0, 1)
1899:     ZEND_ARG_INFO(0, index)
1900: ZEND_END_ARG_INFO()
1901:
1902: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_offsetSet, 0, 0, 2)
1903:     ZEND_ARG_INFO(0, index)
1904:     ZEND_ARG_INFO(0, newval)
1905: ZEND_END_ARG_INFO()
1906:
1907: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_append, 0)
1908:     ZEND_ARG_INFO(0, value)
1909: ZEND_END_ARG_INFO()
1910:
1911: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_seek, 0)
1912:     ZEND_ARG_INFO(0, position)
1913: ZEND_END_ARG_INFO()
1914:
1915: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_exchangeArray, 0)
1916:     ZEND_ARG_INFO(0, array)
1917: ZEND_END_ARG_INFO()
1918:
1919: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_setFlags, 0)
1920:     ZEND_ARG_INFO(0, flags)
1921: ZEND_END_ARG_INFO()
1922:
1923: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_getIteratorClass, 0)
1924:     ZEND_ARG_INFO(0, iteratorClass)
1925: ZEND_END_ARG_INFO()
1926:
1927: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_uksort, 0)
1928:     ZEND_ARG_INFO(0, cmp_function)
1929: ZEND_END_ARG_INFO()
1930:
1931: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_unserialize, 0)
1932:     ZEND_ARG_INFO(0, serialized)
1933: ZEND_END_ARG_INFO()
1934:
1935: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_void, 0)
1936: ZEND_END_ARG_INFO()
1937:
1938: static const zend_function_entry spl_funcs_arrayObject[] = {
1939:     SPL_ME(Array, __construct, arginfo_array___construct, ZEND_ACC_PUBLIC)
1940:     SPL_ME(Array, offsetExists, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1941:     SPL_ME(Array, offsetGet, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1942:     SPL_ME(Array, offsetSet, arginfo_array_offsetSet, ZEND_ACC_PUBLIC)
1943:     SPL_ME(Array, offsetUnset, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1944:     SPL_ME(Array, append, arginfo_array_append, ZEND_ACC_PUBLIC)
1945:     SPL_ME(Array, getArrayCopy, arginfo_array_void, ZEND_ACC_PUBLIC)
1946:     SPL_ME(Array, count, arginfo_array_void, ZEND_ACC_PUBLIC)
1947:     SPL_ME(Array, getFlags, arginfo_array_void, ZEND_ACC_PUBLIC)
1948:     SPL_ME(Array, setFlags, arginfo_array_setFlags, ZEND_ACC_PUBLIC)
1949:     SPL_ME(Array, asort, arginfo_array_void, ZEND_ACC_PUBLIC)
1950:     SPL_ME(Array, usort, arginfo_array_void, ZEND_ACC_PUBLIC)
1951:     SPL_ME(Array, uasort, arginfo_array_uksort, ZEND_ACC_PUBLIC)
1952:     SPL_ME(Array, uksort, arginfo_array_uksort, ZEND_ACC_PUBLIC)
1953:     SPL_ME(Array, natsort, arginfo_array_void, ZEND_ACC_PUBLIC)
1954:     SPL_ME(Array, natcasesort, arginfo_array_void, ZEND_ACC_PUBLIC)
1955:     SPL_ME(Array, unserialize, arginfo_array_unserialize, ZEND_ACC_PUBLIC)
1956:     SPL_ME(Array, serialize, arginfo_array_void, ZEND_ACC_PUBLIC)
1957:     /* ArrayObject specific */
1958:     SPL_ME(Array, getIterator, arginfo_array_void, ZEND_ACC_PUBLIC)
1959:     SPL_ME(Array, exchangeArray, arginfo_array_exchangeArray, ZEND_ACC_PUBLIC)
1960:     SPL_ME(Array, setIteratorClass, arginfo_array_getIteratorClass, ZEND_ACC_PUBLIC)
1961:     SPL_ME(Array, getIteratorClass, arginfo_array_void, ZEND_ACC_PUBLIC)
1962:     PHP_FE_END
1963: };
1964:
1965: static const zend_function_entry spl_funcs_arrayIterator[] = {
1966:     SPL_ME(ArrayIterator, __construct, arginfo_array_iterator___construct, ZEND_ACC_PUBLIC)
1967:     SPL_ME(Array, offsetExists, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1968:     SPL_ME(Array, offsetGet, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1969:     SPL_ME(Array, offsetSet, arginfo_array_offsetSet, ZEND_ACC_PUBLIC)
1970:     SPL_ME(Array, offsetUnset, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1971:     SPL_ME(Array, append, arginfo_array_append, ZEND_ACC_PUBLIC)
1972:     SPL_ME(Array, getArrayCopy, arginfo_array_void, ZEND_ACC_PUBLIC)
1973:     SPL_ME(Array, count, arginfo_array_void, ZEND_ACC_PUBLIC)
1974:     SPL_ME(Array, getFlags, arginfo_array_void, ZEND_ACC_PUBLIC)
1975:     SPL_ME(Array, setFlags, arginfo_array_setFlags, ZEND_ACC_PUBLIC)
1976:     SPL_ME(Array, asort, arginfo_array_void, ZEND_ACC_PUBLIC)
1977:     SPL_ME(Array, usort, arginfo_array_void, ZEND_ACC_PUBLIC)
1978:     SPL_ME(Array, uasort, arginfo_array_uksort, ZEND_ACC_PUBLIC)
1979:     SPL_ME(Array, uksort, arginfo_array_uksort, ZEND_ACC_PUBLIC)
1980:     SPL_ME(Array, natsort, arginfo_array_void, ZEND_ACC_PUBLIC)
1981:     SPL_ME(Array, natcasesort, arginfo_array_void, ZEND_ACC_PUBLIC)
1982:     SPL_ME(Array, unserialize, arginfo_array_unserialize, ZEND_ACC_PUBLIC)
1983:     SPL_ME(Array, serialize, arginfo_array_void, ZEND_ACC_PUBLIC)
1984:     /* ArrayIterator specific */
1985:     SPL_ME(Array, rewind, arginfo_array_void, ZEND_ACC_PUBLIC)
1986:     SPL_ME(Array, current, arginfo_array_void, ZEND_ACC_PUBLIC)
1987:     SPL_ME(Array, key, arginfo_array_void, ZEND_ACC_PUBLIC)
1988:     SPL_ME(Array, next, arginfo_array_void, ZEND_ACC_PUBLIC)
1989:     SPL_ME(Array, valid, arginfo_array_void, ZEND_ACC_PUBLIC)
1990:     SPL_ME(Array, seek, arginfo_array_seek, ZEND_ACC_PUBLIC)
1991:     PHP_FE_END
1992: };
1993:
1994: static const zend_function_entry spl_funcs_recursiveArrayIterator[] = {
1995:     SPL_ME(Array, hasChildren, arginfo_array_void, ZEND_ACC_PUBLIC)
1996:     SPL_ME(Array, getChildren, arginfo_array_void, ZEND_ACC_PUBLIC)
1997:     PHP_FE_END
1998: };
1999: /* }}} */
2000:
2001: /* {{{ PHP_MINIT_FUNCTION(spl_array) */
2002: PHP_MINIT_FUNCTION(spl_array)
2003: {
2004:     REGISTER_SPL_STD_CLASS_EX(ArrayObject, spl_array_object_new, spl_funcs_arrayObject);
2005:     REGISTER_SPL_IMPLEMENTATIONS(ArrayObject, Aggregate);
2006:     REGISTER_SPL_IMPLEMENTATIONS(ArrayObject, ArrayAccess);
2007:     REGISTER_SPL_IMPLEMENTATIONS(ArrayObject, Serializable);
2008:     REGISTER_SPL_IMPLEMENTATIONS(ArrayObject, Countable);
2009:     memcpy(spl_handler_arrayObject, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
2010:
2011:     spl_handler_arrayObject->offset = XTOffsetOf(spl_array_object, std);
2012:
2013:     spl_handler_arrayObject->clone_obj = spl_array_object_clone;
2014:     spl_handler_arrayObject->read_dimension = spl_array_read_dimension;
2015:     spl_handler_arrayObject->write_dimension = spl_array_write_dimension;
2016:     spl_handler_arrayObject->unset_dimension = spl_array_unset_dimension;
2017:     spl_handler_arrayObject->has_dimension = spl_array_has_dimension;
2018:     spl_handler_arrayObject->count_elements = spl_array_object_count_elements;
2019:
2020:     spl_handler_arrayObject->get_properties = spl_array_get_properties;
2021:     spl_handler_arrayObject->get_debug_info = spl_array_get_debug_info;
2022:     spl_handler_arrayObject->get_ptr = spl_array_get_ptr;
2023:     spl_handler_arrayObject->read_property = spl_array_read_property;
2024:     spl_handler_arrayObject->write_property = spl_array_write_property;
2025:     spl_handler_arrayObject->get_ptr_ptr = spl_array_get_ptr_ptr_ptr;
2026:     spl_handler_arrayObject->has_property = spl_array_has_property;
2027:     spl_handler_arrayObject->unset_property = spl_array_unset_property;
2028:
2029:     spl_handler_arrayObject->compare_objects = spl_array_compare_objects;
2030:     spl_handler_arrayObject->dtor_obj = zend_objects_destroy_obj;
2031:     spl_handler_arrayObject->free_obj = spl_array_object_free_storage;
2032:
2033:     REGISTER_SPL_STD_CLASS_EX(ArrayIterator, spl_array_iterator_new, spl_funcs_arrayIterator);
2034:     REGISTER_SPL_IMPLEMENTATIONS(ArrayIterator, Iterator);
2035:     REGISTER_SPL_IMPLEMENTATIONS(ArrayIterator, ArrayAccess);
2036:     REGISTER_SPL_IMPLEMENTATIONS(ArrayIterator, SeekableIterator);
2037:     REGISTER_SPL_IMPLEMENTATIONS(ArrayIterator, Serializable);
2038:     REGISTER_SPL_IMPLEMENTATIONS(ArrayIterator, Countable);
2039:     memcpy(spl_handler_arrayIterator, spl_handler_arrayObject, sizeof(zend_object_handlers));
2040:     spl_array_iterator->get_iterator = spl_array_get_iterator;
2041:
2042:     REGISTER_SPL_CLASS_CONST_LONG(ArrayObject, "STD_PROP_LIST", SPL_ARRAY_STD_PROP_LIST);
2043:     REGISTER_SPL_CLASS_CONST_LONG(ArrayObject, "ARRAY_AS_PROPS", SPL_ARRAY_ARRAY_AS_PROPS);
2044:
2045:     REGISTER_SPL_CLASS_CONST_LONG(ArrayIterator, "STD_PROP_LIST", SPL_ARRAY_STD_PROP_LIST);
2046:     REGISTER_SPL_CLASS_CONST_LONG(ArrayIterator, "ARRAY_AS_PROPS", SPL_ARRAY_ARRAY_AS_PROPS);
2047:
2048:     REGISTER_SPL_SUB_CLASS_EX(RecursiveArrayIterator, ArrayIterator, spl_array_iterator_new, spl_funcs_recursiveArrayIterator);
2049:     REGISTER_SPL_IMPLEMENTATIONS(RecursiveArrayIterator, RecursiveIterator);
2050:     spl_recursiveArrayIterator->get_iterator = spl_array_get_iterator;
2051:
2052:     REGISTER_SPL_CLASS_CONST_LONG(RecursiveArrayIterator, "CHILD_ARRAYS_ONLY", SPL_ARRAY_CHILD_ARRAYS_ONLY);
2053:
2054:     return SUCCESS;
2055: }
2056: /* }}} */
2057:
2058: /*
2059:  * Local Variables:
2060:  * tab-width: 4
2061:  * c-basic-offset: 4
2062:  * End:
2063:  * vim600: fdm=marker
2064:  * vim: noet sw=4 ts=4
2065:  */

```

```
1: /*
2:  *-----*
3:  * | PHP Version 7 |
4:  *-----*
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  *-----*
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  *-----*
15:  * | Authors: Etienne Kneuss <colder@php.net> |
16:  *-----*
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_HEAP_H
22: #define SPL_HEAP_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26:
27: extern PHPAPI zend_class_entry *spl_ce_SplHeap;
28: extern PHPAPI zend_class_entry *spl_ce_SplMinHeap;
29: extern PHPAPI zend_class_entry *spl_ce_SplMaxHeap;
30:
31: extern PHPAPI zend_class_entry *spl_ce_SplPriorityQueue;
32:
33: PHP_MINIT_FUNCTION(spl_heap);
34:
35: #endif /* SPL_HEAP_H */
36:
37: /*
38:  * Local Variables:
39:  * c-basic-offset: 4
40:  * tab-width: 4
41:  * End:
42:  * vim600: fdm=marker
43:  * vim: noet sw=4 ts=4
44:  */
```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_ENGINE_H
22: #define SPL_ENGINE_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26: #include "zend_interfaces.h"
27:
28: #ifndef ZEND_API
29: void spl_instantiate(zend_class_entry *pce, zval *object);
30: #endif
31: zend_long spl_offset_convert_to_long(zval *offset);
32:
33: /* {{{ spl_instantiate_arg_ex1 */
34: static inline int spl_instantiate_arg_ex1(zend_class_entry *pce, zval *retval, zval *arg1)
35: {
36:     zend_function *func = pce->constructor;
37:     spl_instantiate(pce, retval);
38:     zend_call_method(retval, pce, &func, ZEND_VAL(func->common.function_name), ZEND_LEN(func->common.function_name), NULL, 1, arg1, NULL);
39:     return 0;
40: }
41: /* }}} */
42:
43: /* {{{ spl_instantiate_arg_ex2 */
44: static inline int spl_instantiate_arg_ex2(zend_class_entry *pce, zval *retval, zval *arg1, zval *arg2)
45: {
46:     zend_function *func = pce->constructor;
47:     spl_instantiate(pce, retval);
48:     zend_call_method(retval, pce, &func, ZEND_VAL(func->common.function_name), ZEND_LEN(func->common.function_name), NULL, 2, arg1, arg2);
49:     return 0;
50: }
51: /* }}} */
52:
53: /* {{{ spl_instantiate_arg_n */
54: static inline void spl_instantiate_arg_n(zend_class_entry *pce, zval *retval, int argc, zval *argv)
55: {
56:     zend_function *func = pce->constructor;
57:     zend_fcall_info fci;
58:     zend_fcall_info_cache fcc;
59:     zval dummy;
60:
61:     spl_instantiate(pce, retval);
62:
63:     fci.size = sizeof(zend_fcall_info);
64:     ZVAL_STR(&fci.function_name, func->common.function_name);
65:     fci.object = Z_OBJ_P(retval);
66:     fci.retval = &dummy;
67:     fci.param_count = argc;
68:     fci.params = argv;
69:     fci.no_separation = 1;
70:
71:     fcc.function_handler = func;
72:     fcc.calling_scope = zend_get_executed_scope();
73:     fcc.called_scope = pce;
74:     fcc.object = Z_OBJ_P(retval);
75:
76:     zend_call_function(&fci, &fcc);
77:
78: }
79: /* }}} */
80:
81: #endif /* SPL_ENGINE_H */
82:
83: /*
84:  * Local Variables:
85:  * mode: c
86:  * tab-width: 4
87:  * End:
88:  * vim: noet ts=4 ts=4
89:  */
90:
```





```

376:     } else {
377:         zend_clear_exception();
378:     }
379: }
380: }
381: goto next_step;
382: }
383: /* no more elements */
384: if (object->level > 0) {
385:     if (object->endChildren) {
386:         zend_call_method_with_0_params(&this, object->ce, object->zendChildren, "endchildren", NULL);
387:     }
388:     if (EG(exception)) {
389:         if (!object->flags & RIT_CATCH_GET_CHILD) {
390:             return;
391:         } else {
392:             zend_clear_exception();
393:         }
394:     }
395:     if (object->level > 0) {
396:         zval garbage;
397:         ZVAL_COPY_VALUE(&garbage, sub_iter->iterators[object->level].sub_iter);
398:         ZVAL_UNDEF(&object->iterators[object->level].sub_iter);
399:         zval_ptr_dtor(&garbage);
400:         zend_iterator_dtor(iterator);
401:         object->level--;
402:     }
403: } else {
404:     return; /* done completely */
405: }
406: }
407: }
408:
409: static void spl_recursive_it_rewind_ex(spl_recursive_it_object *object, zval *rthis)
410: {
411:     zend_object_iterator *sub_iter;
412:
413:     SPL_FETCH_SUB_ITERATOR(sub_iter, object);
414:
415:     while (object->level) {
416:         sub_iter = object->iterators[object->level].iterator;
417:         zend_iterator_dtor(sub_iter);
418:         zval_ptr_dtor(&object->iterators[object->level-1].sub_iter);
419:         if (EG(exception) && (object->endChildren || object->endChildren->common.scope != spl_ce_RecursiveIteratorIterator)) {
420:             zend_call_method_with_0_params(&this, object->ce, object->zendChildren, "endchildren", NULL);
421:         }
422:     }
423:
424:     object->iterators = erealloc(object->iterators, sizeof(spl_sub_iterator));
425:     object->iterators[0].state = RS_START;
426:     sub_iter = object->iterators[0].iterator;
427:     if (sub_iter->funcs->rewind) {
428:         sub_iter->funcs->rewind(sub_iter);
429:     }
430:
431:     if (EG(exception) && object->beginIteration == 1) {
432:         zend_call_method_with_0_params(&this, object->ce, object->beginIteration, "beginIteration", NULL);
433:     }
434:     object->in_iteration = 1;
435:     spl_recursive_it_move_forward_ex(object, rthis);
436: }
437:
438: static void spl_recursive_it_move_forward(spl_recursive_it_object *iter)
439: {
440:     spl_recursive_it_move_forward_ex(&iter->sub_iter, iter->data);
441: }
442:
443: static void spl_recursive_it_rewind(spl_recursive_it_object *iter)
444: {
445:     spl_recursive_it_rewind_ex(&iter->sub_iter, iter->data);
446: }
447:
448: static zend_object_iterator *spl_recursive_it_get_iterator(spl_recursive_it_object *ce, zval *object, int by_ref)
449: {
450:     spl_recursive_it_iterator *iterator;
451:     spl_recursive_it_object *obj;
452:
453:     if (by_ref) {
454:         zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
455:         return NULL;
456:     }
457:
458:     iterator = emalloc(sizeof(spl_recursive_it_iterator));
459:     object = &iter->sub_iter->sub_iter;
460:     if (object->iterators == NULL) {
461:         zend_error(E_ERROR, "The object to be iterated is in an invalid state: "
462:             "the parent constructor has not been called");
463:     }
464:
465:     zend_iterator_init(&zend_object_iterator) iterator;
466:
467:     ZVAL_COPY(iterator->intern.data, object);
468:     iterator->intern.funcs = ce->iterator_funcs.funcs;
469:     return &zend_object_iterator->iterator;
470: }
471:
472: static const zend_object_iterator_funcs spl_recursive_it_iterator_funcs = {
473:     spl_recursive_it_dtor,
474:     spl_recursive_it_valid,
475:     spl_recursive_it_get_current_data,
476:     spl_recursive_it_get_current_key,
477:     spl_recursive_it_move_forward,
478:     spl_recursive_it_rewind,
479:     NULL
480: };
481:
482: static void spl_recursive_it_it_construct(INTERNAL_FUNCTION_PARAMETERS, zend_class_entry *ce_base, zend_class_entry *ce_inner, recursive_it_type rit_type)
483: {
484:     zval *object = getThis();
485:     spl_recursive_it_object *intern;
486:     zval *iterator;
487:     zend_class_entry *ce_iterator;
488:     zend_long mode, flags;
489:     zend_error_handling error_handling;
490:     zval caching_it, aggregate_retval;
491:
492:     zend_replace_error_handling(EH_THROW, spl_ce_InvalidArgumentException, &error_handling);
493:
494:     switch (rit_type) {
495:         case RIT_RecursiveForAggregate: {
496:             zval caching_it_flags, *user_caching_it_flags = NULL;
497:             mode = RIT_HELP_FIRST;
498:             flags = RIT_BYPASS_RIT;
499:
500:             if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "o|l", iterator, &mode, &flags, user_caching_it_flags, &mode) == SUCCESS) {
501:                 if (instanceof_function(&object->ce, spl_ce_RecursiveForAggregate)) {
502:                     zend_call_method_with_0_params(iterator, &object->ce, iterator->funcs, "if_new_iterator", "getIterator", &aggregate_retval);
503:
504:                     iterator = &aggregate_retval;
505:                 } else {
506:                     if (user_caching_it_flags) {
507:                         ZVAL_COPY(&caching_it_flags, user_caching_it_flags);
508:                     } else {
509:                         ZVAL_LONG(&caching_it_flags, CTT_CATCH_GET_CHILD);
510:                     }
511:                     spl_instantiate_arg_ex2(spl_ce_RecursiveCachingIterator, &caching_it, iterator, &caching_it_flags);
512:                     zval_ptr_dtor(&caching_it_flags);
513:                     zval_ptr_dtor(iterator);
514:                     iterator = &caching_it;
515:                 } else {
516:                     iterator = NULL;
517:                 }
518:                 break;
519:             }
520:             case RIT_RecursiveIteratorForAggregate: {
521:                 default: {
522:                     mode = RIT_LEAVES_ONLY;
523:                     flags = 0;
524:
525:                     if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "o|l", iterator, &mode, &flags) == SUCCESS) {
526:                         if (instanceof_function(&object->ce, spl_ce_RecursiveForAggregate)) {
527:                             zend_call_method_with_0_params(iterator, &object->ce, iterator->funcs, "if_new_iterator", "getIterator", &aggregate_retval);
528:
529:                             iterator = &aggregate_retval;
530:                         } else {
531:                             if (&object->ce != spl_ce_RecursiveForAggregate) {
532:                                 zend_throw_exception(spl_ce_InvalidArgumentException, "An instance of RecursiveIterator or IteratorAggregate creating it is required", 0);
533:                                 zend_restore_error_handling(&error_handling);
534:                                 return;
535:                             }
536:                         }
537:                     }
538:                     if (iterator) {
539:                         if (iterator) {
540:                             zval_ptr_dtor(iterator);
541:                         }
542:                         zend_throw_exception(spl_ce_InvalidArgumentException, "An instance of RecursiveIterator or IteratorAggregate creating it is required", 0);
543:                         zend_restore_error_handling(&error_handling);
544:                         return;
545:                     }
546:
547:                     intern = &object->ce;
548:                     intern->iterators = emalloc(sizeof(spl_sub_iterator));
549:                     intern->level = 0;
550:                     intern->mode = mode;
551:                     intern->flags = (int) flags;
552:                     intern->max_depth = -1;
553:                     intern->in_iteration = 0;
554:                     intern->ce = &object->ce;
555:
556:                     intern->beginIteration = zend_hash_str_find_ptr(intern->ce->function_table, "beginIteration", sizeof("beginIteration") - 1);
557:                     if (intern->beginIteration->common.scope == ce_base) {
558:                         intern->beginIteration = NULL;
559:                     }
560:                     intern->endIteration = zend_hash_str_find_ptr(intern->ce->function_table, "endIteration", sizeof("endIteration") - 1);
561:
562:                     if (intern->endIteration->common.scope == ce_base) {
563:                         intern->endIteration = NULL;
564:                     }
565:
566:                     intern->callHasChildren = zend_hash_str_find_ptr(intern->ce->function_table, "callHasChildren", sizeof("callHasChildren") - 1);
567:                     if (intern->callHasChildren->common.scope == ce_base) {
568:                         intern->callHasChildren = NULL;
569:                     }
570:
571:                     intern->callGetChildren = zend_hash_str_find_ptr(intern->ce->function_table, "callGetChildren", sizeof("callGetChildren") - 1);
572:                     if (intern->callGetChildren->common.scope == ce_base) {
573:                         intern->callGetChildren = NULL;
574:                     }
575:
576:                     intern->beginChildren = zend_hash_str_find_ptr(intern->ce->function_table, "beginChildren", sizeof("beginChildren") - 1);
577:                     if (intern->beginChildren->common.scope == ce_base) {
578:                         intern->beginChildren = NULL;
579:                     }
580:
581:                     intern->endChildren = zend_hash_str_find_ptr(intern->ce->function_table, "endChildren", sizeof("endChildren") - 1);
582:                     if (intern->endChildren->common.scope == ce_base) {
583:                         intern->endChildren = NULL;
584:                     }
585:
586:                     intern->nextElement = zend_hash_str_find_ptr(intern->ce->function_table, "nextElement", sizeof("nextElement") - 1);
587:                     if (intern->nextElement->common.scope == ce_base) {
588:                         intern->nextElement = NULL;
589:                     }
590:
591:                     ce_iterator = &object->ce;
592:                     intern->iterators[0].iterator = ce_iterator->get_iterator(ce_iterator, iterator, 0);
593:                     ZVAL_COPY_VALUE(intern->iterators[0].sub_iter, iterator);
594:                     intern->iterators[0].state = RS_START;
595:
596:                     zend_restore_error_handling(&error_handling);
597:
598:                     if (EG(exception)) {
599:                         zend_object_iterator *sub_iter;
600:
601:                         while (intern->level > 0) {
602:                             sub_iter = intern->iterators[intern->level].iterator;
603:                             zend_iterator_dtor(sub_iter);
604:                             zval_ptr_dtor(&intern->iterators[intern->level-1].sub_iter);
605:                         }
606:                         ifree(intern->iterators);
607:                         intern->iterators = NULL;
608:                     }
609:
610:                     zend_restore_error_handling(&error_handling);
611:
612:                     if (EG(exception)) {
613:                         zend_object_iterator *sub_iter;
614:
615:                         while (intern->level > 0) {
616:                             sub_iter = intern->iterators[intern->level].iterator;
617:                             zend_iterator_dtor(sub_iter);
618:                             zval_ptr_dtor(&intern->iterators[intern->level-1].sub_iter);
619:                         }
620:                         ifree(intern->iterators);
621:                         intern->iterators = NULL;
622:                     }
623:
624:                     /* (( proto void RecursiveIteratorIterator::rewind()
625:                     (( Record the iterator to the first element of the top level inner iterator. */
626:                     SPL_METHOD(RecursiveIteratorIterator, rewind)
627:                     {
628:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
629:
630:                         if (zend_parse_parameters_none() == FAILURE) {
631:                             return;
632:                         }
633:
634:                         spl_recursive_it_rewind_ex(object, getThis());
635:                     }
636:
637:                     /* (( proto bool RecursiveIteratorIterator::valid()
638:                     (( Check whether the current position is valid */
639:                     SPL_METHOD(RecursiveIteratorIterator, valid)
640:                     {
641:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
642:
643:                         if (zend_parse_parameters_none() == FAILURE) {
644:                             return;
645:                         }
646:
647:                         RETURN_BOOL(spl_recursive_it_valid_ex(object, getThis()) == SUCCESS);
648:                     }
649:
650:                     /* (( proto mixed RecursiveIteratorIterator::key()
651:                     (( Access the current key */
652:                     SPL_METHOD(RecursiveIteratorIterator, key)
653:                     {
654:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
655:                         zend_object_iterator *iterator;
656:
657:                         if (zend_parse_parameters_none() == FAILURE) {
658:                             return;
659:                         }
660:
661:                         SPL_FETCH_SUB_ITERATOR(iterator, object);
662:
663:                         if (iterator->funcs->get_current_key) {
664:                             iterator->funcs->get_current_key(iterator, return_value);
665:                         } else {
666:                             RETURN_NULL();
667:                         }
668:                     }
669:
670:                     /* (( proto mixed RecursiveIteratorIterator::current()
671:                     (( Access the current element value */
672:                     SPL_METHOD(RecursiveIteratorIterator, current)
673:                     {
674:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
675:                         zend_object_iterator *iterator;
676:                         zval *data;
677:
678:                         if (zend_parse_parameters_none() == FAILURE) {
679:                             return;
680:                         }
681:
682:                         SPL_FETCH_SUB_ITERATOR(iterator, object);
683:
684:                         data = iterator->funcs->get_current_data(iterator);
685:                         if (data) {
686:                             ZVAL_COPY(&data, data);
687:                         }
688:                     }
689:
690:                     /* (( proto void RecursiveIteratorIterator::next()
691:                     (( Move forward to the next element */
692:                     SPL_METHOD(RecursiveIteratorIterator, next)
693:                     {
694:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
695:
696:                         if (zend_parse_parameters_none() == FAILURE) {
697:                             return;
698:                         }
699:
700:                         spl_recursive_it_move_forward_ex(object, getThis());
701:                     }
702:
703:                     /* (( proto void RecursiveIteratorIterator::getDepth()
704:                     (( Get the current depth of the recursive iteration */
705:                     SPL_METHOD(RecursiveIteratorIterator, getDepth)
706:                     {
707:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
708:
709:                         if (zend_parse_parameters_none() == FAILURE) {
710:                             return;
711:                         }
712:
713:                         RETURN_LONG(object->level);
714:                     }
715:
716:                     /* (( proto RecursiveIteratorIterator::getSubIterator(int level)
717:                     (( The current active sub iterator or the iterator at specified level */
718:                     SPL_METHOD(RecursiveIteratorIterator, getSubIterator)
719:                     {
720:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
721:                         zend_long level = object->level;
722:                         zval *value;
723:
724:                         if (zend_parse_parameters(ZEND_NUM_ARGS(), "[i]", &level) == FAILURE) {
725:                             return;
726:                         }
727:
728:                         if (level < 0 || level > object->level) {
729:                             RETURN_NULL();
730:                         }
731:
732:                         if (object->iterators) {
733:                             zend_throw_exception(spl_ce_LogicException, 0, "The object is in an invalid state as the parent constructor was not called");
734:                             return;
735:                         }
736:
737:                         value = object->iterators[level].sub_iter;
738:                         ZVAL_COPY(&value, value);
739:                         ZVAL_COPY(&return_value, value);
740:                     }
741:
742:                     /* (( proto RecursiveIteratorIterator::getInnerIterator()
743:                     (( The current active sub iterator */
744:                     SPL_METHOD(RecursiveIteratorIterator, getInnerIterator)
745:                     {
746:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
747:                         zval *sub_iter;
748:
749:                         if (zend_parse_parameters_none() == FAILURE) {
750:                             return;
751:                         }
752:
753:                         SPL_FETCH_SUB_ELEMENT_ADDR(sub_iter, object, sub_iter);
754:
755:                         ZVAL_COPY(&sub_iter, sub_iter);
756:                     }
757:
758:                     /* (( proto void RecursiveIteratorIterator::rewind()
759:                     (( Record the iterator to the first element of the top level inner iterator. */
760:                     SPL_METHOD(RecursiveIteratorIterator, rewind)
761:                     {
762:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
763:
764:                         if (zend_parse_parameters_none() == FAILURE) {
765:                             return;
766:                         }
767:
768:                         spl_recursive_it_rewind_ex(object, getThis());
769:                     }
770:
771:                     /* (( proto bool RecursiveIteratorIterator::valid()
772:                     (( Check whether the current position is valid */
773:                     SPL_METHOD(RecursiveIteratorIterator, valid)
774:                     {
775:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
776:
777:                         if (zend_parse_parameters_none() == FAILURE) {
778:                             return;
779:                         }
780:
781:                         RETURN_BOOL(spl_recursive_it_valid_ex(object, getThis()) == SUCCESS);
782:                     }
783:
784:                     /* (( proto mixed RecursiveIteratorIterator::key()
785:                     (( Access the current key */
786:                     SPL_METHOD(RecursiveIteratorIterator, key)
787:                     {
788:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
789:                         zend_object_iterator *iterator;
790:
791:                         if (zend_parse_parameters_none() == FAILURE) {
792:                             return;
793:                         }
794:
795:                         SPL_FETCH_SUB_ITERATOR(iterator, object);
796:
797:                         if (iterator->funcs->get_current_key) {
798:                             iterator->funcs->get_current_key(iterator, return_value);
799:                         } else {
800:                             RETURN_NULL();
801:                         }
802:                     }
803:
804:                     /* (( proto mixed RecursiveIteratorIterator::current()
805:                     (( Access the current element value */
806:                     SPL_METHOD(RecursiveIteratorIterator, current)
807:                     {
808:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
809:                         zend_object_iterator *iterator;
810:                         zval *data;
811:
812:                         if (zend_parse_parameters_none() == FAILURE) {
813:                             return;
814:                         }
815:
816:                         SPL_FETCH_SUB_ITERATOR(iterator, object);
817:
818:                         data = iterator->funcs->get_current_data(iterator);
819:                         if (data) {
820:                             ZVAL_COPY(&data, data);
821:                         }
822:                     }
823:
824:                     /* (( proto void RecursiveIteratorIterator::next()
825:                     (( Move forward to the next element */
826:                     SPL_METHOD(RecursiveIteratorIterator, next)
827:                     {
828:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
829:
830:                         if (zend_parse_parameters_none() == FAILURE) {
831:                             return;
832:                         }
833:
834:                         spl_recursive_it_move_forward_ex(object, getThis());
835:                     }
836:
837:                     /* (( proto void RecursiveIteratorIterator::getDepth()
838:                     (( Get the current depth of the recursive iteration */
839:                     SPL_METHOD(RecursiveIteratorIterator, getDepth)
840:                     {
841:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
842:
843:                         if (zend_parse_parameters_none() == FAILURE) {
844:                             return;
845:                         }
846:
847:                         RETURN_LONG(object->level);
848:                     }
849:
850:                     /* (( proto RecursiveIteratorIterator::getSubIterator(int level)
851:                     (( The current active sub iterator or the iterator at specified level */
852:                     SPL_METHOD(RecursiveIteratorIterator, getSubIterator)
853:                     {
854:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
855:
856:                         if (zend_parse_parameters(ZEND_NUM_ARGS(), "[i]", &level) == FAILURE) {
857:                             return;
858:                         }
859:
860:                         if (level < 0 || level > object->level) {
861:                             RETURN_NULL();
862:                         }
863:
864:                         if (object->iterators) {
865:                             zend_throw_exception(spl_ce_LogicException, 0, "The object is in an invalid state as the parent constructor was not called");
866:                             return;
867:                         }
868:
869:                         value = object->iterators[level].sub_iter;
870:                         ZVAL_COPY(&value, value);
871:                         ZVAL_COPY(&return_value, value);
872:                     }
873:
874:                     /* (( proto RecursiveIteratorIterator::getInnerIterator()
875:                     (( The current active sub iterator */
876:                     SPL_METHOD(RecursiveIteratorIterator, getInnerIterator)
877:                     {
878:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
879:                         zval *sub_iter;
880:
881:                         if (zend_parse_parameters_none() == FAILURE) {
882:                             return;
883:                         }
884:
885:                         SPL_FETCH_SUB_ELEMENT_ADDR(sub_iter, object, sub_iter);
886:
887:                         ZVAL_COPY(&sub_iter, sub_iter);
888:                     }
889:
890:                     /* (( proto void RecursiveIteratorIterator::rewind()
891:                     (( Record the iterator to the first element of the top level inner iterator. */
892:                     SPL_METHOD(RecursiveIteratorIterator, rewind)
893:                     {
894:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
895:
896:                         if (zend_parse_parameters_none() == FAILURE) {
897:                             return;
898:                         }
899:
900:                         spl_recursive_it_rewind_ex(object, getThis());
901:                     }
902:
903:                     /* (( proto bool RecursiveIteratorIterator::valid()
904:                     (( Check whether the current position is valid */
905:                     SPL_METHOD(RecursiveIteratorIterator, valid)
906:                     {
896:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
897:
898:                         if (zend_parse_parameters_none() == FAILURE) {
899:                             return;
900:                         }
901:
902:                         RETURN_BOOL(spl_recursive_it_valid_ex(object, getThis()) == SUCCESS);
903:                     }
904:
905:                     /* (( proto mixed RecursiveIteratorIterator::key()
906:                     (( Access the current key */
907:                     SPL_METHOD(RecursiveIteratorIterator, key)
908:                     {
909:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
910:                         zend_object_iterator *iterator;
911:
912:                         if (zend_parse_parameters_none() == FAILURE) {
913:                             return;
914:                         }
915:
916:                         SPL_FETCH_SUB_ITERATOR(iterator, object);
917:
918:                         if (iterator->funcs->get_current_key) {
919:                             iterator->funcs->get_current_key(iterator, return_value);
920:                         } else {
921:                             RETURN_NULL();
922:                         }
923:                     }
924:
925:                     /* (( proto mixed RecursiveIteratorIterator::current()
926:                     (( Access the current element value */
927:                     SPL_METHOD(RecursiveIteratorIterator, current)
928:                     {
929:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
930:                         zend_object_iterator *iterator;
931:                         zval *data;
932:
933:                         if (zend_parse_parameters_none() == FAILURE) {
934:                             return;
935:                         }
936:
937:                         SPL_FETCH_SUB_ITERATOR(iterator, object);
938:
939:                         data = iterator->funcs->get_current_data(iterator);
940:                         if (data) {
941:                             ZVAL_COPY(&data, data);
942:                         }
943:                     }
944:
945:                     /* (( proto void RecursiveIteratorIterator::next()
946:                     (( Move forward to the next element */
947:                     SPL_METHOD(RecursiveIteratorIterator, next)
948:                     {
949:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
950:
951:                         if (zend_parse_parameters_none() == FAILURE) {
952:                             return;
953:                         }
954:
955:                         spl_recursive_it_move_forward_ex(object, getThis());
956:                     }
957:
958:                     /* (( proto void RecursiveIteratorIterator::getDepth()
959:                     (( Get the current depth of the recursive iteration */
960:                     SPL_METHOD(RecursiveIteratorIterator, getDepth)
961:                     {
962:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
963:
964:                         if (zend_parse_parameters_none() == FAILURE) {
965:                             return;
966:                         }
967:
968:                         RETURN_LONG(object->level);
969:                     }
970:
971:                     /* (( proto RecursiveIteratorIterator::getSubIterator(int level)
972:                     (( The current active sub iterator or the iterator at specified level */
973:                     SPL_METHOD(RecursiveIteratorIterator, getSubIterator)
974:                     {
975:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
976:
977:                         if (zend_parse_parameters(ZEND_NUM_ARGS(), "[i]", &level) == FAILURE) {
978:                             return;
979:                         }
980:
981:                         if (level < 0 || level > object->level) {
982:                             RETURN_NULL();
983:                         }
984:
985:                         if (object->iterators) {
986:                             zend_throw_exception(spl_ce_LogicException, 0, "The object is in an invalid state as the parent constructor was not called");
987:                             return;
988:                         }
989:
990:                         value = object->iterators[level].sub_iter;
991:                         ZVAL_COPY(&value, value);
992:                         ZVAL_COPY(&return_value, value);
993:                     }
994:
995:                     /* (( proto RecursiveIteratorIterator::getInnerIterator()
996:                     (( The current active sub iterator */
997:                     SPL_METHOD(RecursiveIteratorIterator, getInnerIterator)
998:                     {
999:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1000:                         zval *sub_iter;
1001:
1002:                         if (zend_parse_parameters_none() == FAILURE) {
1003:                             return;
1004:                         }
1005:
1006:                         SPL_FETCH_SUB_ELEMENT_ADDR(sub_iter, object, sub_iter);
1007:
1008:                         ZVAL_COPY(&sub_iter, sub_iter);
1009:                     }
1010:
1011:                     /* (( proto void RecursiveIteratorIterator::rewind()
1012:                     (( Record the iterator to the first element of the top level inner iterator. */
1013:                     SPL_METHOD(RecursiveIteratorIterator, rewind)
1014:                     {
1015:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1016:
1017:                         if (zend_parse_parameters_none() == FAILURE) {
1018:                             return;
1019:                         }
1020:
1021:                         spl_recursive_it_rewind_ex(object, getThis());
1022:                     }
1023:
1024:                     /* (( proto bool RecursiveIteratorIterator::valid()
1025:                     (( Check whether the current position is valid */
1026:                     SPL_METHOD(RecursiveIteratorIterator, valid)
1027:                     {
1028:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1029:
1030:                         if (zend_parse_parameters_none() == FAILURE) {
1031:                             return;
1032:                         }
1033:
1034:                         RETURN_BOOL(spl_recursive_it_valid_ex(object, getThis()) == SUCCESS);
1035:                     }
1036:
1037:                     /* (( proto mixed RecursiveIteratorIterator::key()
1038:                     (( Access the current key */
1039:                     SPL_METHOD(RecursiveIteratorIterator, key)
1040:                     {
1041:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1042:                         zend_object_iterator *iterator;
1043:
1044:                         if (zend_parse_parameters_none() == FAILURE) {
1045:                             return;
1046:                         }
1047:
1048:                         SPL_FETCH_SUB_ITERATOR(iterator, object);
1049:
1050:                         if (iterator->funcs->get_current_key) {
1051:                             iterator->funcs->get_current_key(iterator, return_value);
1052:                         } else {
1053:                             RETURN_NULL();
1054:                         }
1055:                     }
1056:
1057:                     /* (( proto mixed RecursiveIteratorIterator::current()
1058:                     (( Access the current element value */
1059:                     SPL_METHOD(RecursiveIteratorIterator, current)
1060:                     {
1061:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1062:                         zend_object_iterator *iterator;
1063:                         zval *data;
1064:
1065:                         if (zend_parse_parameters_none() == FAILURE) {
1066:                             return;
1067:                         }
1068:
1069:                         SPL_FETCH_SUB_ITERATOR(iterator, object);
1070:
1071:                         data = iterator->funcs->get_current_data(iterator);
1072:                         if (data) {
1073:                             ZVAL_COPY(&data, data);
1074:                         }
1075:                     }
1076:
1077:                     /* (( proto void RecursiveIteratorIterator::next()
1078:                     (( Move forward to the next element */
1079:                     SPL_METHOD(RecursiveIteratorIterator, next)
1080:                     {
1081:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1082:
1083:                         if (zend_parse_parameters_none() == FAILURE) {
1084:                             return;
1085:                         }
1086:
1087:                         spl_recursive_it_move_forward_ex(object, getThis());
1088:                     }
1089:
1090:                     /* (( proto void RecursiveIteratorIterator::getDepth()
1091:                     (( Get the current depth of the recursive iteration */
1092:                     SPL_METHOD(RecursiveIteratorIterator, getDepth)
1093:                     {
1094:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1095:
1096:                         if (zend_parse_parameters_none() == FAILURE) {
1097:                             return;
1098:                         }
1099:
1100:                         RETURN_LONG(object->level);
1101:                     }
1102:
1103:                     /* (( proto RecursiveIteratorIterator::getSubIterator(int level)
1104:                     (( The current active sub iterator or the iterator at specified level */
1105:                     SPL_METHOD(RecursiveIteratorIterator, getSubIterator)
1106:                     {
1107:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1108:
1109:                         if (zend_parse_parameters(ZEND_NUM_ARGS(), "[i]", &level) == FAILURE) {
1110:                             return;
1111:                         }
1112:
1113:                         if (level < 0 || level > object->level) {
1114:                             RETURN_NULL();
1115:                         }
1116:
1117:                         if (object->iterators) {
1118:                             zend_throw_exception(spl_ce_LogicException, 0, "The object is in an invalid state as the parent constructor was not called");
1119:                             return;
1120:                         }
1121:
1122:                         value = object->iterators[level].sub_iter;
1123:                         ZVAL_COPY(&value, value);
1124:                         ZVAL_COPY(&return_value, value);
1125:                     }
1126:
1127:                     /* (( proto RecursiveIteratorIterator::getInnerIterator()
1128:                     (( The current active sub iterator */
1129:                     SPL_METHOD(RecursiveIteratorIterator, getInnerIterator)
1130:                     {
1131:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1132:                         zval *sub_iter;
1133:
1134:                         if (zend_parse_parameters_none() == FAILURE) {
1135:                             return;
1136:                         }
1137:
1138:                         SPL_FETCH_SUB_ELEMENT_ADDR(sub_iter, object, sub_iter);
1139:
1140:                         ZVAL_COPY(&sub_iter, sub_iter);
1141:                     }
1142:
1143:                     /* (( proto void RecursiveIteratorIterator::rewind()
1144:                     (( Record the iterator to the first element of the top level inner iterator. */
1145:                     SPL_METHOD(RecursiveIteratorIterator, rewind)
1146:                     {
1147:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1148:
1149:                         if (zend_parse_parameters_none() == FAILURE) {
1150:                             return;
1151:                         }
1152:
1153:                         spl_recursive_it_rewind_ex(object, getThis());
1154:                     }
1155:
1156:                     /* (( proto bool RecursiveIteratorIterator::valid()
1157:                     (( Check whether the current position is valid */
1158:                     SPL_METHOD(RecursiveIteratorIterator, valid)
1159:                     {
1160:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1161:
1162:                         if (zend_parse_parameters_none() == FAILURE) {
1163:                             return;
1164:                         }
1165:
1166:                         RETURN_BOOL(spl_recursive_it_valid_ex(object, getThis()) == SUCCESS);
1167:                     }
1168:
1169:                     /* (( proto mixed RecursiveIteratorIterator::key()
1170:                     (( Access the current key */
1171:                     SPL_METHOD(RecursiveIteratorIterator, key)
1172:                     {
1173:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1174:                         zend_object_iterator *iterator;
1175:
1176:                         if (zend_parse_parameters_none() == FAILURE) {
1177:                             return;
1178:                         }
1179:
1180:                         SPL_FETCH_SUB_ITERATOR(iterator, object);
1181:
1182:                         if (iterator->funcs->get_current_key) {
1183:                             iterator->funcs->get_current_key(iterator, return_value);
1184:                         } else {
1185:                             RETURN_NULL();
1186:                         }
1187:                     }
1188:
1189:                     /* (( proto mixed RecursiveIteratorIterator::current()
1190:                     (( Access the current element value */
1191:                     SPL_METHOD(RecursiveIteratorIterator, current)
1192:                     {
1193:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1194:                         zend_object_iterator *iterator;
1195:                         zval *data;
1196:
1197:                         if (zend_parse_parameters_none() == FAILURE) {
1198:                             return;
1199:                         }
1200:
1201:                         SPL_FETCH_SUB_ITERATOR(iterator, object);
1202:
1203:                         data = iterator->funcs->get_current_data(iterator);
1204:                         if (data) {
1205:                             ZVAL_COPY(&data, data);
1206:                         }
1207:                     }
1208:
1209:                     /* (( proto void RecursiveIteratorIterator::next()
1210:                     (( Move forward to the next element */
1211:                     SPL_METHOD(RecursiveIteratorIterator, next)
1212:                     {
1213:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1214:
1215:                         if (zend_parse_parameters_none() == FAILURE) {
1216:                             return;
1217:                         }
1218:
1219:                         spl_recursive_it_move_forward_ex(object, getThis());
1220:                     }
1221:
1222:                     /* (( proto void RecursiveIteratorIterator::getDepth()
1223:                     (( Get the current depth of the recursive iteration */
1224:                     SPL_METHOD(RecursiveIteratorIterator, getDepth)
1225:                     {
1226:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1227:
1228:                         if (zend_parse_parameters_none() == FAILURE) {
1229:                             return;
1230:                         }
1231:
1232:                         RETURN_LONG(object->level);
1233:                     }
1234:
1235:                     /* (( proto RecursiveIteratorIterator::getSubIterator(int level)
1236:                     (( The current active sub iterator or the iterator at specified level */
1237:                     SPL_METHOD(RecursiveIteratorIterator, getSubIterator)
1238:                     {
1239:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1240:
1241:                         if (zend_parse_parameters(ZEND_NUM_ARGS(), "[i]", &level) == FAILURE) {
1242:                             return;
1243:                         }
1244:
1245:                         if (level < 0 || level > object->level) {
1246:                             RETURN_NULL();
1247:                         }
1248:
1249:                         if (object->iterators) {
1250:                             zend_throw_exception(spl_ce_LogicException, 0, "The object is in an invalid state as the parent constructor was not called");
1251:                             return;
1252:                         }
1253:
1254:                         value = object->iterators[level].sub_iter;
1255:                         ZVAL_COPY(&value, value);
1256:                         ZVAL_COPY(&return_value, value);
1257:                     }
1258:
1259:                     /* (( proto RecursiveIteratorIterator::getInnerIterator()
1260:                     (( The current active sub iterator */
1261:                     SPL_METHOD(RecursiveIteratorIterator, getInnerIterator)
1262:                     {
1263:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1264:                         zval *sub_iter;
1265:
1266:                         if (zend_parse_parameters_none() == FAILURE) {
1267:                             return;
1268:                         }
1269:
1270:                         SPL_FETCH_SUB_ELEMENT_ADDR(sub_iter, object, sub_iter);
1271:
1272:                         ZVAL_COPY(&sub_iter, sub_iter);
1273:                     }
1274:
1275:                     /* (( proto void RecursiveIteratorIterator::rewind()
1276:                     (( Record the iterator to the first element of the top level inner iterator. */
1277:                     SPL_METHOD(RecursiveIteratorIterator, rewind)
1278:                     {
1279:                         spl_recursive_it_object *object = &intern->sub_iter->sub_iter;
1280:
1281:                         if (zend_parse_parameters_none() == FAILURE) {
1282:                             return;
1283:                         }
1284:
1285:                         spl_recursive_it_rewind_ex(object, getThis());
1286:                     }
1287:
1288:                     /* (( proto bool RecursiveIteratorIterator::valid()
1289:                     ((
```

```

748: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::beginIteration()
749: Called when iteration begins (after first rewind() call) */
750: SPL_METHOD(RecursiveIteratorIterator, beginIteration)
751: {
752:     IF (zend_parse_parameters_none() == FAILURE) {
753:         return;
754:     }
755:     /* nothing to do */
756:     /* }}} */
757: }
758:
759: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::endIteration()
760: Called when iteration ends (when valid() first returns false) */
761: SPL_METHOD(RecursiveIteratorIterator, endIteration)
762: {
763:     IF (zend_parse_parameters_none() == FAILURE) {
764:         return;
765:     }
766:     /* nothing to do */
767:     /* }}} */
768: }
769: /* {{{ proto bool RecursiveIteratorIterator::callHasChildren()
770: Called for each element to test whether it has children */
771: SPL_METHOD(RecursiveIteratorIterator, callHasChildren)
772: {
773:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
774:     zend_class_entry *ce;
775:     zval *obj;
776:
777:     IF (zend_parse_parameters_none() == FAILURE) {
778:         return;
779:     }
780:
781:     IF (obj->iterators) {
782:         RETURN_NULL();
783:     }
784:
785:     SPL_FETCH_SUB_ELEMENT(ce, object, ce);
786:
787:     obj = obj->iterators[object->level].obj;
788:     IF (Z_TYPE_P(obj) == IS_UNDEF) {
789:         RETURN_FALSE;
790:     } else {
791:         zend_call_method_with_0_params(obj, ce, NULL, "haschildren", return_value);
792:         IF (Z_TYPE_P(return_value) == IS_UNDEF) {
793:             RETURN_FALSE;
794:         }
795:     }
796:     /* }}} */
797: }
798: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::callGetChildren()
799: Return children of current element */
800: SPL_METHOD(RecursiveIteratorIterator, callGetChildren)
801: {
802:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
803:     zend_class_entry *ce;
804:     zval *obj;
805:
806:     IF (zend_parse_parameters_none() == FAILURE) {
807:         return;
808:     }
809:
810:     SPL_FETCH_SUB_ELEMENT(ce, object, ce);
811:
812:     obj = obj->iterators[object->level].obj;
813:     IF (Z_TYPE_P(obj) == IS_UNDEF) {
814:         return;
815:     } else {
816:         zend_call_method_with_0_params(obj, ce, NULL, "getchildren", return_value);
817:         IF (Z_TYPE_P(return_value) == IS_UNDEF) {
818:             RETURN_NULL();
819:         }
820:     }
821:     /* }}} */
822: }
823: /* {{{ proto void RecursiveIteratorIterator::beginChildren()
824: Called when recursing one level down */
825: SPL_METHOD(RecursiveIteratorIterator, beginChildren)
826: {
827:     IF (zend_parse_parameters_none() == FAILURE) {
828:         return;
829:     }
830:     /* nothing to do */
831:     /* }}} */
832: }
833: /* {{{ proto void RecursiveIteratorIterator::endChildren()
834: Called when end recursing one level */
835: SPL_METHOD(RecursiveIteratorIterator, endChildren)
836: {
837:     IF (zend_parse_parameters_none() == FAILURE) {
838:         return;
839:     }
840:     /* nothing to do */
841:     /* }}} */
842: }
833: /* {{{ proto void RecursiveIteratorIterator::nextElement()
834: Called when the next element is available */
845: SPL_METHOD(RecursiveIteratorIterator, nextElement)
846: {
847:     IF (zend_parse_parameters_none() == FAILURE) {
848:         return;
849:     }
850:     /* nothing to do */
851:     /* }}} */
852: }
853: /* {{{ proto void RecursiveIteratorIterator::setMaxDepth([max_depth = -1])
854: Set the maximum allowed depth (or any depth if max_depth = -1) */
855: SPL_METHOD(RecursiveIteratorIterator, setMaxDepth)
856: {
857:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
858:     zend_long max_depth = -1;
859:
860:     IF (zend_parse_parameters(ZEND_NUM_ARGS(), "|l", &max_depth) == FAILURE) {
861:         return;
862:     }
863:     IF (max_depth < -1) {
864:         zend_throw_exception(spl_ce_OutOfRangeException, "Parameter max_depth must be >= -1", 0);
865:         return;
866:     } else IF (max_depth > INT_MAX) {
867:         max_depth = INT_MAX;
868:     }
869:
870:     object->max_depth = (int)max_depth;
871:     /* }}} */
872: }
873: /* {{{ proto int|false RecursiveIteratorIterator::getMaxDepth()
874: Return the maximum accepted depth or false if any depth is allowed */
875: SPL_METHOD(RecursiveIteratorIterator, getMaxDepth)
876: {
877:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
878:
879:     IF (zend_parse_parameters_none() == FAILURE) {
880:         return;
881:     }
882:
883:     IF (object->max_depth == -1) {
884:         RETURN_FALSE;
885:     } else {
886:         RETURN_LONG(object->max_depth);
887:     }
888:     /* }}} */
889: }
890:
891: static union zend_function *spl_recursive_it_get_method(zend_object **obj, zend_string *method, const zval *key)
892: {
893:     union_zend_function *function_handler;
894:     spl_recursive_it_object *object = spl_recursive_it_from_obj(*obj);
895:     zend_long level = object->level;
896:     zval *obj;
897:
898:     IF (obj->iterators) {
899:         php_error_docref(NULL, E_WARNING, "The instance wasn't initialized properly", ZSTR_VAL(*obj), "ce->name");
900:     }
901:     obj = obj->iterators[level].obj;
902:     function_handler = std_object_handlers.get_method(obj, method, key);
903:     IF (function_handler) {
904:         IF (function_handler == zend_hash_find_ptr(&Z_OBJ_P(obj)->function_table, method) == NULL) {
905:             IF (Z_OBJ_HT_P(obj)->get_method) {
906:                 *obj = Z_OBJ_P(obj);
907:                 function_handler = (*obj->handlers)->get_method(obj, method, key);
908:             }
909:             /* }}} */
910:             *obj = Z_OBJ_P(obj);
911:         }
912:     }
913:     return function_handler;
914: }
915:
916: /* {{{ spl_recursive_iterator_iterator_ctor */
917: static void spl_recursive_iterator_iterator_ctor(zend_object **obj)
918: {
919:     spl_recursive_it_object *object = spl_recursive_it_from_obj(*obj);
920:     zend_object_iterator *sub_iter;
921:
922:     /* call standard ctor */
923:     zend_objects_destroy_object(*obj);
924:
925:     IF (obj->iterators) {
926:         while (object->level >= 0) {
927:             sub_iter = obj->iterators[object->level].iterator;
928:             zend_iterator_ctor(sub_iter);
929:             zval_ptr_dtor(&obj->iterators[object->level--].obj);
930:         }
931:         efree(obj->iterators);
932:         obj->iterators = NULL;
933:     }
934:     /* }}} */
935: }
936:
937: /* {{{ spl_recursive_iterator_iterator_free_storage */
938: static void spl_recursive_iterator_iterator_free_storage(zend_object **obj)
939: {
940:     spl_recursive_it_object *object = spl_recursive_it_from_obj(*obj);
941:
942:     IF (obj->iterators) {
943:         efree(obj->iterators);
944:         obj->iterators = NULL;
945:         object->level = 0;
946:     }
947:
948:     zend_object_iterator_dtor(obj->level);
949:     smart_str_free(obj->prefix[0]);
950:     smart_str_free(obj->prefix[1]);
951:     smart_str_free(obj->prefix[2]);
952:     smart_str_free(obj->prefix[3]);
953:     smart_str_free(obj->prefix[4]);
954:     smart_str_free(obj->prefix[5]);
955:
956:     smart_str_free(obj->postfix[0]);
957: }
958: /* }}} */
959:
960: /* {{{ spl_recursive_iterator_iterator_new */
961: static zend_object *spl_recursive_iterator_iterator_new(zend_class_entry *class_type, int init_prefix)
962: {
963:     spl_recursive_it_object *intern;
964:
965:     intern = zend_object_alloc(sizeof(spl_recursive_it_object), class_type);
966:
967:     IF (init_prefix) {
968:         smart_str_append(&intern->prefix[0], "", 0);
969:         smart_str_append(&intern->prefix[1], "", 2);
970:         smart_str_append(&intern->prefix[2], "", 2);
971:         smart_str_append(&intern->prefix[3], "", 2);
972:         smart_str_append(&intern->prefix[4], "\\", 2);
973:         smart_str_append(&intern->prefix[5], "", 0);
974:
975:         smart_str_append(&intern->postfix[0], "", 0);
976:     }
977:
978:     zend_object_iterator_init(&intern->std, class_type);
979:     object_properties_init(&intern->std, class_type);
980:
981:     intern->std.handlers = &spl_handlers_rec_it;
982:     return intern->std;
983: }
984: /* }}} */
985:
986: /* {{{ spl_recursive_iterator_iterator_new */
987: static zend_object *spl_recursive_iterator_iterator_new(zend_class_entry *class_type)
988: {
989:     return spl_recursive_iterator_iterator_new(class_type, 0);
990: }
991: /* }}} */
992:
993: /* {{{ spl_recursive_iterator_iterator_new */
994: static zend_object *spl_recursive_iterator_iterator_new(zend_class_entry *class_type)
995: {
996:     return spl_recursive_iterator_iterator_new(class_type, 1);
997: }
998: /* }}} */
999:
1000: ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_it_construct, 0, 0, 1)
1001:     ZEND_ARG_OBJ_INFO(0, iterator, Traversable, 0)
1002:     ZEND_ARG_INFO(0, mode)
1003:     ZEND_ARG_INFO(0, flags)
1004: ZEND_END_ARG_INFO();
1005:
1006: ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_it_getSubIterator, 0, 0, 0)
1007:     ZEND_ARG_INFO(0, level)
1008: ZEND_END_ARG_INFO();
1009:
1010: ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_it_setMaxDepth, 0, 0, 0)
1011:     ZEND_ARG_INFO(0, max_depth)
1012: ZEND_END_ARG_INFO();
1013:
1014: static const zend_function_entry spl_funcs_recursive_iterator[] = {
1015:     SPL_ME(RecursiveIteratorIterator, __construct, arginfo_recursive_it_construct, ZEND_ACC_PUBLIC),
1016:     SPL_ME(RecursiveIteratorIterator, rewind, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
1017:     SPL_ME(RecursiveIteratorIterator, valid, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
1018:     SPL_ME(RecursiveIteratorIterator, key, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
1019:     SPL_ME(RecursiveIteratorIterator, current, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
1020:     SPL_ME(RecursiveIteratorIterator, next, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
1021:     SPL_ME(RecursiveIteratorIterator, getDepth, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
1022:     SPL_ME(RecursiveIteratorIterator, getSubIterator, arginfo_recursive_it_getSubIterator, ZEND_ACC_PUBLIC),
1023:     SPL_ME(RecursiveIteratorIterator, getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
1024:     SPL_ME(RecursiveIteratorIterator, beginIteration, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
1025:     SPL_ME(RecursiveIteratorIterator, endIteration, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
1026:     SPL_ME(RecursiveIteratorIterator, callHasChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
1027:     SPL_ME(RecursiveIteratorIterator, callGetChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
1028:     SPL_ME(RecursiveIteratorIterator, beginChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
1029:     SPL_ME(RecursiveIteratorIterator, nextElement, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
1030:     SPL_ME(RecursiveIteratorIterator, setMaxDepth, arginfo_recursive_it_setMaxDepth, ZEND_ACC_PUBLIC),
1031:     SPL_ME(RecursiveIteratorIterator, getMaxDepth, arginfo_recursive_it_getMaxDepth, ZEND_ACC_PUBLIC),
1032:     PHP_FPE_END
1033: };
1034:
1035:
1036: static void spl_recursive_tree_iterator_get_prefix(spl_recursive_it_object *object, zval *return_value)
1037: {
1038:     smart_str str = {0};
1039:     zval has_next;
1040:     int level;
1041:
1042:     smart_str_append(&str, ZSTR_VAL(object->prefix[0].s), ZSTR_LEN(object->prefix[0].s);
1043:
1044:     for (level = 0; level < object->level+1; level++) {
1045:         zend_call_method_with_0_params(object->iterators[level].obj, object->iterators[level].ce, NULL, "hasnext", &has_next);
1046:         IF (Z_TYPE(has_next) != IS_UNDEF) {
1047:             IF (Z_TYPE(has_next) == IS_TRUE) {
1048:                 smart_str_append(&str, ZSTR_VAL(object->prefix[1].s), ZSTR_LEN(object->prefix[1].s);
1049:             } else {
1050:                 smart_str_append(&str, ZSTR_VAL(object->prefix[2].s), ZSTR_LEN(object->prefix[2].s);
1051:             }
1052:             zval_ptr_dtor(&has_next);
1053:         }
1054:
1055:         zend_call_method_with_0_params(object->iterators[level].obj, object->iterators[level].ce, NULL, "hasnext", &has_next);
1056:         IF (Z_TYPE(has_next) != IS_UNDEF) {
1057:             IF (Z_TYPE(has_next) == IS_TRUE) {
1058:                 smart_str_append(&str, ZSTR_VAL(object->prefix[3].s), ZSTR_LEN(object->prefix[3].s);
1059:             } else {
1060:                 smart_str_append(&str, ZSTR_VAL(object->prefix[4].s), ZSTR_LEN(object->prefix[4].s);
1061:             }
1062:             zval_ptr_dtor(&has_next);
1063:         }
1064:
1065:         smart_str_append(&str, ZSTR_VAL(object->prefix[5].s), ZSTR_LEN(object->prefix[5].s);
1066:         smart_str_0(&str);
1067:
1068:         RETURN_NEW_STR(str.s);
1069:     }
1070:
1071:     static void spl_recursive_tree_iterator_get_entry(spl_recursive_it_object *object, zval *return_value)
1072: {
1073:     zend_object_iterator *iterator = object->iterators[object->level].iterator;
1074:     zval *data;
1075:     zend_error_handling error_handling;
1076:
1077:     data = iterator->funcs->get_current_data(iterator);
1078:
1079:     /* Replace exception handling so the catchable fatal error that is thrown when a class
1080:     * without __toString is converted to string is converted to an exception. */
1081:     zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, error_handling);
1082:     IF (data) {
1083:         ZVAL_DEREF(data);
1084:         IF (Z_TYPE_P(data) == IS_ARRAY) {
1085:             ZVAL_STRING(return_value, "Array", sizeof("Array")-1);
1086:         } else {
1087:             ZVAL_COPY(return_value, data);
1088:             convert_to_string(return_value);
1089:         }
1090:     }
1091:
1092:     zend_restore_error_handling(error_handling);
1093:
1094:     static void spl_recursive_tree_iterator_get_postfix(spl_recursive_it_object *object, zval *return_value)
1095: {
1096:     RETVAL_STR(object->postfix[0].s);
1097:     Z_ADDREF_P(return_value);
1098: }
1099:
1100: /* {{{ proto void RecursiveTreeIterator::__construct(RecursiveIteratorIterator $iterator, int flags = RTT_BYPASS_KEY, int cit_flags = CIT_CATCH_ON_CHILD, int mode = RTT_STOP_STOP) */
1101: RecursiveTreeIterator to generate ASCII graphic trees for the entries in a RecursiveIterator */
1102: SPL_METHOD(RecursiveTreeIterator, __construct)
1103: {
1104:     spl_recursive_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveTreeIterator, zend_ce_iterator, RTT_RecursiveTreeIterator);
1105:     /* }}} */
1106:
1107: /* {{{ proto void RecursiveTreeIterator::setPrefixPart(int part, string prefix) throws OutOfRangeException */
1108: Set prefix parts as used in getPrefix() */
1109: SPL_METHOD(RecursiveTreeIterator, setPrefixPart)
1110: {
1111:     zend_long part;
1112:     char *prefix;
1113:     size_t prefix_len;
1114:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1115:
1116:     IF (zend_parse_parameters(ZEND_NUM_ARGS(), "is", &part, &prefix, &prefix_len) == FAILURE) {
1117:         return;
1118:     }
1119:
1120:     IF (0 > part || part > 5) {
1121:         zend_throw_exception(spl_ce_OutOfRangeException, 0, "Use RecursiveTreeIterator::PREFIX_* constant");
1122:         return;
1123:     }

```

```

1124: smart_str_free(object->prefix[part]);
1125:
1126: smart_str_append(object->prefix[part], prefix, prefix_len);
1127: /* '}' */
1128:
1129: /* {{{ proto string RecursiveTreeIterator::getPrefix()
1130:  Returns the string to place in front of current element */
1131: SPL_METHOD (RecursiveTreeIterator, getPrefix)
1132: {
1133:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1134:
1135:     if (zend_parse_parameters_none() == FAILURE) {
1136:         return;
1137:     }
1138:
1139:     if(!object->itersators) {
1140:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1141:             "The object is in an invalid state as the parent constructor was not called");
1142:         return;
1143:     }
1144:
1145:     spl_recursive_tree_iterator_get_prefix(object, return_value);
1146:     /* '}' */
1147:
1148:     /* {{{ proto void RecursiveTreeIterator::setPostfix(string prefix)
1149:      Sets postfix as used in getPrefix() */
1150: SPL_METHOD (RecursiveTreeIterator, setPostfix)
1151: {
1152:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1153:     char* postfix;
1154:     size_t postfix_len;
1155:
1156:     if (zend_parse_parameters(ZEND_NUM_ARGS() & "s", &postfix, &postfix_len) == FAILURE) {
1157:         return;
1158:     }
1159:
1160:     smart_str_free(object->postfix[0]);
1161:     smart_str_append(object->postfix[0], postfix, postfix_len);
1162:     /* '}' */
1163:
1164:     /* {{{ proto string RecursiveTreeIterator::getEntry()
1165:  Returns the string presentation built for current element */
1166: SPL_METHOD (RecursiveTreeIterator, getEntry)
1167: {
1168:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1169:
1170:     if (zend_parse_parameters_none() == FAILURE) {
1171:         return;
1172:     }
1173:
1174:     if(!object->itersators) {
1175:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1176:             "The object is in an invalid state as the parent constructor was not called");
1177:         return;
1178:     }
1179:
1180:     spl_recursive_tree_iterator_get_entry(object, return_value);
1181:     /* '}' */
1182:
1183:     /* {{{ proto string RecursiveTreeIterator::getPostfix()
1184:  Returns the string to place after the current element */
1185: SPL_METHOD (RecursiveTreeIterator, getPostfix)
1186: {
1187:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1188:
1189:     if (zend_parse_parameters_none() == FAILURE) {
1190:         return;
1191:     }
1192:
1193:     if(!object->itersators) {
1194:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1195:             "The object is in an invalid state as the parent constructor was not called");
1196:         return;
1197:     }
1198:
1199:     spl_recursive_tree_iterator_get_postfix(object, return_value);
1200:     /* '}' */
1201:
1202:     /* {{{ proto mixed RecursiveTreeIterator::current()
1203:  Returns the current element prefixed and postfixed */
1204: SPL_METHOD (RecursiveTreeIterator, current)
1205: {
1206:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1207:     zval prefix, entry, postfix;
1208:     char *ptr;
1209:     zend_string *str;
1210:
1211:     if (zend_parse_parameters_none() == FAILURE) {
1212:         return;
1213:     }
1214:
1215:     if(!object->itersators) {
1216:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1217:             "The object is in an invalid state as the parent constructor was not called");
1218:         return;
1219:     }
1220:
1221:     if (object->flags & RTIT_BYPASS_CURRENT) {
1222:         zend_object_iterator *iterator = object->itersators[object->level].iterator;
1223:         zval *data;
1224:
1225:         SPL_FETCH_SUB_ITERATOR(iterator, object);
1226:         data = iterator->funcs->get_current_data(iterator);
1227:         if (data) {
1228:             ZVAL_DEREF(data);
1229:             ZVAL_COPY(return_value, data);
1230:             return;
1231:         } else {
1232:             RETURN_NULL();
1233:         }
1234:     }
1235:
1236:     ZVAL_NULL(&prefix);
1237:     ZVAL_NULL(&entry);
1238:     spl_recursive_tree_iterator_get_prefix(object, &prefix);
1239:     spl_recursive_tree_iterator_get_entry(object, &entry);
1240:     if (Z_TYPE(entry) != IS_STRING) {
1241:         zval_ptr_dtor(&entry);
1242:         zval_ptr_dtor(&prefix);
1243:         RETURN_NULL();
1244:     }
1245:
1246:     spl_recursive_tree_iterator_get_postfix(object, &postfix);
1247:
1248:     str = zend_string_alloc(Z_STRLEN(prefix) + Z_STRLEN(entry) + Z_STRLEN(postfix), 0);
1249:     ptr = ZSTR_VAL(str);
1250:
1251:     memcpy(ptr, Z_STRVAL(prefix), Z_STRLEN(prefix));
1252:     ptr += Z_STRLEN(prefix);
1253:     memcpy(ptr, Z_STRVAL(entry), Z_STRLEN(entry));
1254:     ptr += Z_STRLEN(entry);
1255:     memcpy(ptr, Z_STRVAL(postfix), Z_STRLEN(postfix));
1256:     ptr += Z_STRLEN(postfix);
1257:     *ptr = 0;
1258:
1259:     zval_ptr_dtor(&prefix);
1260:     zval_ptr_dtor(&entry);
1261:     zval_ptr_dtor(&postfix);
1262:
1263:     RETURN_NEW_STR(str);
1264:     /* '}' */
1265:
1266:     /* {{{ proto mixed RecursiveTreeIterator::key()
1267:  Returns the current key prefixed and postfixed */
1268: SPL_METHOD (RecursiveTreeIterator, key)
1269: {
1270:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1271:     zend_object_iterator *iterator;
1272:     zval prefix, key, postfix, key_copy;
1273:     char *ptr;
1274:     zend_string *str;
1275:
1276:     if (zend_parse_parameters_none() == FAILURE) {
1277:         return;
1278:     }
1279:
1280:     SPL_FETCH_SUB_ITERATOR(iterator, object);
1281:
1282:     if (iterator->funcs->get_current_key() {
1283:         iterator->funcs->get_current_key(iterator, &key);
1284:     } else {
1285:         ZVAL_NULL(&key);
1286:     }
1287:
1288:     if (object->flags & RTIT_BYPASS_KEY) {
1289:         RETVAL_ZVAL(&key, 1, 1);
1290:         return;
1291:     }
1292:
1293:     if (Z_TYPE(key) != IS_STRING) {
1294:         if (zend_make_printable_zval(&key, &key_copy)) {
1295:             key = key_copy;
1296:         }
1297:     }
1298:
1299:     spl_recursive_tree_iterator_get_prefix(object, &prefix);
1300:     spl_recursive_tree_iterator_get_postfix(object, &postfix);
1301:
1302:     str = zend_string_alloc(Z_STRLEN(prefix) + Z_STRLEN(key) + Z_STRLEN(postfix), 0);
1303:     ptr = ZSTR_VAL(str);
1304:
1305:     memcpy(ptr, Z_STRVAL(prefix), Z_STRLEN(prefix));
1306:     ptr += Z_STRLEN(prefix);
1307:     memcpy(ptr, Z_STRVAL(key), Z_STRLEN(key));
1308:     ptr += Z_STRLEN(key);
1309:     memcpy(ptr, Z_STRVAL(postfix), Z_STRLEN(postfix));
1310:     ptr += Z_STRLEN(postfix);
1311:     *ptr = 0;
1312:
1313:     zval_ptr_dtor(&prefix);
1314:     zval_ptr_dtor(&key);
1315:     zval_ptr_dtor(&postfix);
1316:
1317:     RETURN_NEW_STR(str);
1318:     /* '}' */
1319:
1320:     /* {{{ proto void RecursiveTreeIterator::rewind()
1321:  Rewind the iterator to the first element */
1322: SPL_METHOD (RecursiveTreeIterator, rewind)
1323: {
1324:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1325:
1326:     if (zend_parse_parameters_none() == FAILURE) {
1327:         return;
1328:     }
1329:
1330:     if(!object->itersators) {
1331:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1332:             "The object is in an invalid state as the parent constructor was not called");
1333:         return;
1334:     }
1335:
1336:     spl_recursive_tree_iterator_rewind(object);
1337:     /* '}' */
1338:
1339:     /* {{{ proto void RecursiveTreeIterator::valid()
1340:  Checks if the iterator is valid (not exhausted) */
1341: SPL_METHOD (RecursiveTreeIterator, valid)
1342: {
1343:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1344:
1345:     if (zend_parse_parameters_none() == FAILURE) {
1346:         return;
1347:     }
1348:
1349:     if(!object->itersators) {
1350:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1351:             "The object is in an invalid state as the parent constructor was not called");
1352:         return;
1353:     }
1354:
1355:     spl_recursive_tree_iterator_valid(object);
1356:     /* '}' */
1357:
1358:     /* {{{ proto void RecursiveTreeIterator::current()
1359:  Returns the current element */
1360: SPL_METHOD (RecursiveTreeIterator, current)
1361: {
1362:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1363:
1364:     if (zend_parse_parameters_none() == FAILURE) {
1365:         return;
1366:     }
1367:
1368:     if(!object->itersators) {
1369:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1370:             "The object is in an invalid state as the parent constructor was not called");
1371:         return;
1372:     }
1373:
1374:     spl_recursive_tree_iterator_current(object);
1375:     /* '}' */
1376:
1377:     /* {{{ proto void RecursiveTreeIterator::next()
1378:  Move the iterator to the next element */
1379: SPL_METHOD (RecursiveTreeIterator, next)
1380: {
1381:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1382:
1383:     if (zend_parse_parameters_none() == FAILURE) {
1384:         return;
1385:     }
1386:
1387:     if(!object->itersators) {
1388:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1389:             "The object is in an invalid state as the parent constructor was not called");
1390:         return;
1391:     }
1392:
1393:     spl_recursive_tree_iterator_next(object);
1394:     /* '}' */
1395:
1396:     /* {{{ proto void RecursiveTreeIterator::end()
1397:  Move the iterator to the end of the tree */
1398: SPL_METHOD (RecursiveTreeIterator, end)
1399: {
1400:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1401:
1402:     if (zend_parse_parameters_none() == FAILURE) {
1403:         return;
1404:     }
1405:
1406:     if(!object->itersators) {
1407:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1408:             "The object is in an invalid state as the parent constructor was not called");
1409:         return;
1410:     }
1411:
1412:     spl_recursive_tree_iterator_end(object);
1413:     /* '}' */
1414:
1415:     /* {{{ proto void RecursiveTreeIterator::callasChildren()
1416:  Call the iterator on the children of the current node */
1417: SPL_METHOD (RecursiveTreeIterator, callasChildren)
1418: {
1419:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1420:
1421:     if (zend_parse_parameters_none() == FAILURE) {
1422:         return;
1423:     }
1424:
1425:     if(!object->itersators) {
1426:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1427:
```

```

1489: zend_throw_exception(spl_ce_invalidArgumentException, "Flags must contain only one of CALL_TO_STRING, TO_STRING_USE_KEY, TO_STRING_USE_CURRENT, TO
1490:   STRING_USE_INDEX", 0);
1491: return NULL;
1492: }
1493: }
1494: }
1495: }
1496: }
1497: }
1498: }
1499: }
1500: }
1501: }
1502: }
1503: }
1504: }
1505: }
1506: }
1507: }
1508: }
1509: }
1510: }
1511: }
1512: }
1513: }
1514: }
1515: }
1516: }
1517: }
1518: }
1519: }
1520: }
1521: }
1522: }
1523: }
1524: }
1525: }
1526: }
1527: }
1528: }
1529: }
1530: }
1531: }
1532: }
1533: }
1534: }
1535: }
1536: }
1537: }
1538: }
1539: }
1540: }
1541: }
1542: }
1543: }
1544: }
1545: }
1546: }
1547: }
1548: }
1549: }
1550: }
1551: }
1552: }
1553: }
1554: }
1555: }
1556: }
1557: }
1558: }
1559: }
1560: }
1561: }
1562: }
1563: }
1564: }
1565: }
1566: }
1567: }
1568: }
1569: }
1570: }
1571: }
1572: }
1573: }
1574: }
1575: }
1576: }
1577: }
1578: }
1579: }
1580: }
1581: }
1582: }
1583: }
1584: }
1585: }
1586: }
1587: }
1588: }
1589: }
1590: }
1591: }
1592: }
1593: }
1594: }
1595: }
1596: }
1597: }
1598: }
1599: }
1600: }
1601: }
1602: }
1603: }
1604: }
1605: }
1606: }
1607: }
1608: }
1609: }
1610: }
1611: }
1612: }
1613: }
1614: }
1615: }
1616: }
1617: }
1618: }
1619: }
1620: }
1621: }
1622: }
1623: }
1624: }
1625: }
1626: }
1627: }
1628: }
1629: }
1630: }
1631: }
1632: }
1633: }
1634: }
1635: }
1636: }
1637: }
1638: }
1639: }
1640: }
1641: }
1642: }
1643: }
1644: }
1645: }
1646: }
1647: }
1648: }
1649: }
1650: }
1651: }
1652: }
1653: }
1654: }
1655: }
1656: }
1657: }
1658: }
1659: }
1660: }
1661: }
1662: }
1663: }
1664: }
1665: }
1666: }
1667: }
1668: }
1669: }
1670: }
1671: }
1672: }
1673: }
1674: }
1675: }
1676: }
1677: }
1678: }
1679: }
1680: }
1681: }
1682: }
1683: }
1684: }
1685: }
1686: }
1687: }
1688: }
1689: }
1690: }
1691: }
1692: }
1693: }
1694: }
1695: }
1696: }
1697: }
1698: }
1699: }
1700: }
1701: }
1702: }
1703: }
1704: }
1705: }
1706: }
1707: }
1708: }
1709: }
1710: }
1711: }
1712: }
1713: }
1714: }
1715: }
1716: }
1717: }
1718: }
1719: }
1720: }
1721: }
1722: }
1723: }
1724: }
1725: }
1726: }
1727: }
1728: }
1729: }
1730: }
1731: }
1732: }
1733: }
1734: }
1735: }
1736: }
1737: }
1738: }
1739: }
1740: }
1741: }
1742: }
1743: }
1744: }
1745: }
1746: }
1747: }
1748: }
1749: }
1750: }
1751: }
1752: }
1753: }
1754: }
1755: }
1756: }
1757: }
1758: }
1759: }
1760: }
1761: }
1762: }
1763: }
1764: }
1765: }
1766: }
1767: }
1768: }
1769: }
1770: }
1771: }
1772: }
1773: }
1774: }
1775: }
1776: }
1777: }
1778: }
1779: }
1780: }
1781: }
1782: }
1783: }
1784: }
1785: }
1786: }
1787: }
1788: }
1789: }
1790: }
1791: }
1792: }
1793: }
1794: }
1795: }
1796: }
1797: }
1798: }
1799: }
1800: }
1801: }
1802: }
1803: }
1804: }
1805: }
1806: }
1807: }
1808: }
1809: }
1810: }
1811: }
1812: }
1813: }
1814: }
1815: }
1816: }
1817: }
1818: }
1819: }
1820: }
1821: }
1822: }
1823: }
1824: }
1825: }
1826: }
1827: }
1828: }
1829: }
1830: }
1831: }
1832: }
1833: }
1834: }
1835: }
1836: }
1837: }
1838: }
1839: }
1840: }
1841: }
1842: }
1843: }
1844: }
1845: }
1846: }
1847: }
1848: }
1849: }
1850: }
1851: }
1852: }
1853: }
1854: }
1855: }
1856: }
1857: }
1858: }
1859: }
1860: }
1861: }
1862: }
1863: }
1864: }
1865: }
1866: }
1867: }
1868: }
1869: }
1870: }

```

```

1683: }
1684: }
1685: }
1686: }
1687: }
1688: }
1689: }
1690: }
1691: }
1692: }
1693: }
1694: }
1695: }
1696: }
1697: }
1698: }
1699: }
1700: }
1701: }
1702: }
1703: }
1704: }
1705: }
1706: }
1707: }
1708: }
1709: }
1710: }
1711: }
1712: }
1713: }
1714: }
1715: }
1716: }
1717: }
1718: }
1719: }
1720: }
1721: }
1722: }
1723: }
1724: }
1725: }
1726: }
1727: }
1728: }
1729: }
1730: }
1731: }
1732: }
1733: }
1734: }
1735: }
1736: }
1737: }
1738: }
1739: }
1740: }
1741: }
1742: }
1743: }
1744: }
1745: }
1746: }
1747: }
1748: }
1749: }
1750: }
1751: }
1752: }
1753: }
1754: }
1755: }
1756: }
1757: }
1758: }
1759: }
1760: }
1761: }
1762: }
1763: }
1764: }
1765: }
1766: }
1767: }
1768: }
1769: }
1770: }
1771: }
1772: }
1773: }
1774: }
1775: }
1776: }
1777: }
1778: }
1779: }
1780: }
1781: }
1782: }
1783: }
1784: }
1785: }
1786: }
1787: }
1788: }
1789: }
1790: }
1791: }
1792: }
1793: }
1794: }
1795: }
1796: }
1797: }
1798: }
1799: }
1800: }
1801: }
1802: }
1803: }
1804: }
1805: }
1806: }
1807: }
1808: }
1809: }
1810: }
1811: }
1812: }
1813: }
1814: }
1815: }
1816: }
1817: }
1818: }
1819: }
1820: }
1821: }
1822: }
1823: }
1824: }
1825: }
1826: }
1827: }
1828: }
1829: }
1830: }
1831: }
1832: }
1833: }
1834: }
1835: }
1836: }
1837: }
1838: }
1839: }
1840: }
1841: }
1842: }
1843: }
1844: }
1845: }
1846: }
1847: }
1848: }
1849: }
1850: }
1851: }
1852: }
1853: }
1854: }
1855: }
1856: }
1857: }
1858: }
1859: }
1860: }
1861: }
1862: }
1863: }
1864: }
1865: }
1866: }
1867: }
1868: }
1869: }
1870: }

```

```

1871:     spl_filter_it_fetch(this, intern);
1872: }
1873:
1874: /* {{{ proto void FilterIterator::rewind()
1875:  * Rewind the iterator */
1876: SPL_METHOD(FilterIterator, rewind)
1877: {
1878:     spl_dual_it_object *intern;
1879:
1880:     IF_SEND_PARAM_PARAMETERS_NONE() == FAILURE() {
1881:         return;
1882:     }
1883:
1884:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1885:     spl_filter_it_rewind(getThis(), intern);
1886:     /* }}} */
1887:
1888: /* {{{ proto void FilterIterator::next()
1889:  * Move the iterator forward */
1890: SPL_METHOD(FilterIterator, next)
1891: {
1892:     spl_dual_it_object *intern;
1893:
1894:     IF_SEND_PARAM_PARAMETERS_NONE() == FAILURE() {
1895:         return;
1896:     }
1897:
1898:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1899:     spl_filter_it_next(getThis(), intern);
1900:     /* }}} */
1901:
1902: /* {{{ proto void RecursiveCallbackFilterIterator::__construct(RecursiveIterator it, callback func)
1903:  * Create a RecursiveCallbackFilterIterator from a RecursiveIterator */
1904: SPL_METHOD(RecursiveCallbackFilterIterator, __construct)
1905: {
1906:     spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveCallbackFilterIterator, spl_ca_RecursiveIterator, DIT_RecursiveCallbackFilter
Iterator);
1907:     /* }}} */
1908:
1909:
1910: /* {{{ proto void RecursiveFilterIterator::__construct(RecursiveIterator it)
1911:  * Create a RecursiveFilterIterator from a RecursiveIterator */
1912: SPL_METHOD(RecursiveFilterIterator, __construct)
1913: {
1914:     spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveFilterIterator, spl_ca_RecursiveIterator, DIT_RecursiveFilterIterator);
1915:     /* }}} */
1916:
1917: /* {{{ proto bool RecursiveFilterIterator::hasChildren()
1918:  * Check whether the inner iterator's current element has children */
1919: SPL_METHOD(RecursiveFilterIterator, hasChildren)
1920: {
1921:     spl_dual_it_object *intern;
1922:     zval retval;
1923:
1924:     IF_SEND_PARAM_PARAMETERS_NONE() == FAILURE() {
1925:         return;
1926:     }
1927:
1928:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1929:
1930:     zend_call_method_with_0_params(&intern->inner.object, intern->inner.ce, NULL, "haschildren", &retval);
1931:     IF (Z_TYPE(retval) != IS_UNDEF) {
1932:         RETURN_ZVAL(&retval, 0, 1);
1933:     } else {
1934:         RETURN_FALSE;
1935:     }
1936:     /* }}} */
1937:
1938: /* {{{ proto RecursiveFilterIterator RecursiveFilterIterator::getChildren()
1939:  * Return the inner iterator's children contained in a RecursiveFilterIterator */
1940: SPL_METHOD(RecursiveFilterIterator, getChildren)
1941: {
1942:     spl_dual_it_object *intern;
1943:     zval retval;
1944:
1945:     IF_SEND_PARAM_PARAMETERS_NONE() == FAILURE() {
1946:         return;
1947:     }
1948:
1949:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1950:
1951:     zend_call_method_with_0_params(&intern->inner.object, intern->inner.ce, NULL, "getchildren", &retval);
1952:     IF (EG(exception) && Z_TYPE(retval) != IS_UNDEF) {
1953:         spl_instantiate_arg_ext(Z_OBJCE_P(getThis()), return_value, &retval);
1954:     }
1955:     zend_get_zfor(&retval);
1956:     /* }}} */
1957:
1958: /* {{{ proto RecursiveCallbackFilterIterator RecursiveCallbackFilterIterator::getChildren()
1959:  * Return the inner iterator's children contained in a RecursiveCallbackFilterIterator */
1960: SPL_METHOD(RecursiveCallbackFilterIterator, getChildren)
1961: {
1962:     spl_dual_it_object *intern;
1963:     zval retval;
1964:
1965:     IF_SEND_PARAM_PARAMETERS_NONE() == FAILURE() {
1966:         return;
1967:     }
1968:
1969:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1970:
1971:     zend_call_method_with_0_params(&intern->inner.object, intern->inner.ce, NULL, "getchildren", &retval);
1972:     IF (EG(exception) && Z_TYPE(retval) != IS_UNDEF) {
1973:         spl_instantiate_arg_ext(Z_OBJCE_P(getThis()), return_value, &retval, &intern->cbfilter->fci.function_name);
1974:     }
1975:     zend_get_zfor(&retval);
1976:     /* }}} */
1977:
1978: /* {{{ proto void ParentIterator::__construct(RecursiveIterator it)
1979:  * Create a ParentIterator from a RecursiveIterator */
1980: SPL_METHOD(ParentIterator, __construct)
1981: {
1982:     spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_ParentIterator, spl_ca_RecursiveIterator, DIT_ParentIterator);
1983:     /* }}} */
1984:
1985: /* {{{ proto void RegexIterator::__construct(Iterator it, string regex [, int mode [, int flags [, int preg_flags]])
1986:  * Create an RegexIterator from another iterator and a regular expression */
1987: SPL_METHOD(RegexIterator, __construct)
1988: {
1989:     spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RegexIterator, zend_ca_iterator, DIT_RegexIterator);
1990:     /* }}} */
1991:
1992: /* {{{ proto bool CallbackFilterIterator::accept()
1993:  * Call the callback with the current value, the current key and the inner iterator as arguments */
1994: SPL_METHOD(CallbackFilterIterator, accept)
1995: {
1996:     spl_dual_it_object *intern = Z_SPDUAL_IT_P(getThis());
1997:     zend_fcall_info *fci = &intern->u.cbfilter->fci;
1998:     zend_fcall_info_cache *foc = &intern->u.cbfilter->foc;
1999:     zval params[3];
2000:
2001:     IF_SEND_PARAM_PARAMETERS_NONE() == FAILURE() {
2002:         return;
2003:     }
2004:
2005:     IF (Z_TYPE(intern->current.data) == IS_UNDEF || Z_TYPE(intern->current.key) == IS_UNDEF) {
2006:         RETURN_FALSE;
2007:     }
2008:
2009:     ZVAL_COPY_VALUE(&params[0], &intern->current.data);
2010:     ZVAL_COPY_VALUE(&params[1], &intern->current.key);
2011:     ZVAL_COPY_VALUE(&params[2], &intern->inner.object);
2012:
2013:     fci->retval = return_value;
2014:     fci->param_count = 3;
2015:     fci->params = params;
2016:     fci->no_separation = 0;
2017:
2018:     IF (zend_call_function(fci, foc) != SUCCESS || Z_ISUNDEF(return_value)) {
2019:         RETURN_FALSE;
2020:     }
2021:
2022:     IF (EG(exception)) {
2023:         RETURN_NULL();
2024:     }
2025:
2026:     /* zend_call_function may change args to IS_REF */
2027:     ZVAL_COPY_VALUE(&intern->current.data, &params[0]);
2028:     ZVAL_COPY_VALUE(&intern->current.key, &params[1]);
2029:     /* }}} */
2030:
2031:
2032: /* {{{ proto bool RegexIterator::accept()
2033:  * Match (string)current() against regular expression */
2034: SPL_METHOD(RegexIterator, accept)
2035: {
2036:     spl_dual_it_object *intern;
2037:     zend_string *result, *subject;
2038:     zval count = 0;
2039:     zval account, *replacement, tmp_replacement, rv;
2040:     pcre_match_data *match_data;
2041:     pcre_code *re;
2042:     int rv;
2043:
2044:     IF_SEND_PARAM_PARAMETERS_NONE() == FAILURE() {
2045:         return;
2046:     }
2047:
2048:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2049:
2050:     IF (Z_TYPE(intern->current.data) == IS_UNDEF) {
2051:         RETURN_FALSE;
2052:     }
2053:
2054:     IF (intern->u.regex.flags & REGIT_USE_KEY) {
2055:         subject = zval_get_string(&intern->current.key);
2056:     } else {
2057:         IF (Z_TYPE(intern->current.data) == IS_ARRAY) {
2058:             RETURN_FALSE;
2059:         }
2060:
2061:         RETURN_FALSE;
2062:     }
2063:
2064:     switch (intern->u.regex.mode) {
2065:         case REGIT_MODE_MAX: /* won't happen but makes compiler happy */
2066:         case REGIT_MODE_MATCH:
2067:             re = pcre_get_pcre_re(intern->u.regex.pcre);
2068:             match_data = pcre_create_match_data(0, re);
2069:             IF (!match_data) {
2070:                 RETURN_FALSE;
2071:             }
2072:             rv = pcre2_match(re, (PCRE2_SPTR)ZSTR_VAL(subject), ZSTR_LEN(subject), 0, 0, match_data, pcre_pcre_match());
2073:             RETVAL_BOOL(rv > 0);
2074:             pcre_free(match_data(match_data));
2075:             break;
2076:
2077:         case REGIT_MODE_ALL_MATCHES:
2078:         case REGIT_MODE_GET_MATCH:
2079:             ZVAL_UNDEF(&intern->current.data);
2080:             ZVAL_UNDEF(&intern->current.data);
2081:             pcre_match_impl(intern->u.regex.pcre, ZSTR_VAL(subject), ZSTR_LEN(subject), &account,
2082:                 &intern->current.data, &intern->u.regex.mode == REGIT_MODE_ALL_MATCHES, &intern->u.regex.use_flags, &intern->u.regex.preg_flags, 0);
2083:             RETVAL_BOOL(&Z_STRLEN(account) > 0);
2084:             break;
2085:
2086:         case REGIT_MODE_SPLIT:
2087:             zval_get_zfor(&intern->current.data);
2088:             ZVAL_UNDEF(&intern->current.data);
2089:             pcre_split_impl(intern->u.regex.pcre, subject, &intern->current.data, -1, &intern->u.regex.preg_flags);
2090:             count = zend_hash_num_elements(Z_ARRVAL(intern->current.data));
2091:             RETVAL_BOOL(count > 1);
2092:             break;
2093:
2094:         case REGIT_MODE_REPLACE:
2095:             replacement = zend_read_property(intern->std.ce, getThis(), "replacement", sizeof("replacement")-1, 1, &rv);
2096:             IF (Z_TYPE_P(replacement) != IS_STRING) {
2097:                 ZVAL_COPY(&tmp_replacement, replacement);
2098:                 convert_to_string(&tmp_replacement);
2099:                 replacement = &tmp_replacement;
2100:             }
2101:             result = pcre_replace_impl(intern->u.regex.pcre, subject, ZSTR_VAL(subject), ZSTR_LEN(subject), ZSTR_P(replacement), -1, &account);
2102:
2103:             IF (intern->u.regex.flags & REGIT_USE_KEY) {
2104:                 zval_get_zfor(&intern->current.key);
2105:                 ZVAL_STR(&intern->current.key, result);
2106:             } else {
2107:                 zval_get_zfor(&intern->current.data);
2108:                 ZVAL_STR(&intern->current.data, result);
2109:             }
2110:
2111:             IF (replacement == &tmp_replacement) {
2112:                 zval_get_zfor(&replacement);
2113:             }
2114:             RETVAL_BOOL(count > 0);
2115:         }
2116:
2117:         IF (intern->u.regex.flags & REGIT_INVERTED) {
2118:             RETVAL_BOOL(Z_TYPE_P(return_value) != IS_TRUE);
2119:         }
2120:         zend_string_release(subject);
2121:     } /* }}} */
2122:
2123: /* {{{ proto string RegexIterator::getRegex()
2124:  * Return current regular expression */
2125: SPL_METHOD(RegexIterator, getRegex)
2126: {
2127:     spl_dual_it_object *intern = Z_SPDUAL_IT_P(getThis());
2128:
2129:     IF (zend_param_parameters_none() == FAILURE) {
2130:         return;
2131:     }
2132:
2133:     RETURN_STR_COPY(&intern->u.regex.regex);
2134:     /* }}} */
2135:
2136: /* {{{ proto bool RegexIterator::getMode()
2137:  * Return current operation mode */
2138: SPL_METHOD(RegexIterator, getMode)
2139: {
2140:     spl_dual_it_object *intern;
2141:
2142:     IF (zend_param_parameters_none() == FAILURE) {
2143:         return;
2144:     }
2145:
2146:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2147:
2148:     RETURN_LONG(intern->u.regex.mode);
2149:     /* }}} */
2150:
2151: /* {{{ proto bool RegexIterator::setMode(int new_mode)
2152:  * Set new operation mode */
2153: SPL_METHOD(RegexIterator, setMode)
2154: {
2155:     spl_dual_it_object *intern;
2156:     zend_long mode;
2157:
2158:     IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &mode) == FAILURE) {
2159:         return;
2160:     }
2161:
2162:     IF (mode < 0 || mode >= REGIT_MODE_MAX) {
2163:         zend_throw_exception_ext(spl_ca_invalidArgumentException, 0, "Illegal mode " ZEND_LONG_FMT, mode);
2164:         return; /* NULL */
2165:     }
2166:
2167:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2168:
2169:     intern->u.regex.mode = mode;
2170:     /* }}} */
2171:
2172: /* {{{ proto bool RegexIterator::getFlags()
2173:  * Return current operation flags */
2174: SPL_METHOD(RegexIterator, getFlags)
2175: {
2176:     spl_dual_it_object *intern;
2177:
2178:     IF (zend_param_parameters_none() == FAILURE) {
2179:         return;
2180:     }
2181:
2182:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2183:
2184:     RETURN_LONG(intern->u.regex.flags);
2185:     /* }}} */
2186:
2187: /* {{{ proto bool RegexIterator::setFlags(int new_flags)
2188:  * Set operation flags */
2189: SPL_METHOD(RegexIterator, setFlags)
2190: {
2191:     spl_dual_it_object *intern;
2192:     zend_long flags;
2193:
2194:     IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &flags) == FAILURE) {
2195:         return;
2196:     }
2197:
2198:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2199:
2200:     intern->u.regex.flags = flags;
2201:     /* }}} */
2202:
2203: /* {{{ proto bool RegexIterator::getPregFlags()
2204:  * Return current PREG flags (if in use or NULL) */
2205: SPL_METHOD(RegexIterator, getPregFlags)
2206: {
2207:     spl_dual_it_object *intern;
2208:
2209:     IF (zend_param_parameters_none() == FAILURE) {
2210:         return;
2211:     }
2212:
2213:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2214:
2215:     IF (intern->u.regex.use_flags) {
2216:         RETURN_LONG(intern->u.regex.preg_flags);
2217:     } else {
2218:         return;
2219:     }
2220:     /* }}} */
2221:
2222: /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2223:  * Set PREG flags */
2224: SPL_METHOD(RegexIterator, setPregFlags)
2225: {
2226:     spl_dual_it_object *intern;
2227:     zend_long preg_flags;
2228:
2229:     IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2230:         return;
2231:     }
2232:
2233:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2234:
2235:     intern->u.regex.preg_flags = preg_flags;
2236:     intern->u.regex.use_flags = 1;
2237:     /* }}} */
2238:
2239: /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]])
2240:  * Create a RecursiveRegexIterator from another recursive iterator and a regular expression */
2241: SPL_METHOD(RecursiveRegexIterator, __construct)
2242: {
2243:     spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2244:     /* }}} */
2245:

```



```

2247: // Return the inner iterator's children contained in a RecursiveHashIterator.
2248: SPL_METHOD(RecursiveHashIterator, getChildren)
2249: {
2250:     spl_dual_iterator *intern;
2251:     zval retval;
2252:
2253:     if (zend_parse_parameters_none() == FAILURE) {
2254:         return;
2255:     }
2256:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2257:
2258:     zend_call_method_with_0_params(&intern->inner_obj, intern->inner_ce, NULL, "getChildren", &retval);
2259:     if (IS_EXCEPTION) {
2260:         zval args[5];
2261:         ZVAL_COPY(&args[0], &retval);
2262:         ZVAL_COPY(&args[1], intern->u.regex);
2263:         ZVAL_COPY(&args[2], intern->u.regex_mode);
2264:         ZVAL_COPY(&args[3], intern->u.regex_flags);
2265:         ZVAL_COPY(&args[4], intern->u.regex_preg_flags);
2266:     }
2267:     spl_instantiate_arginfo(INTERNAL_FUNCTION_PARAM_PASSTHRU, return_value, 5, args);
2268:
2269:     zval_ptr_stor(&obj->inner_obj);
2270:     zval_ptr_stor(&obj->inner_obj);
2271:     zval_ptr_stor(&obj->inner_obj);
2272:     zval_ptr_stor(&obj->inner_obj);
2273:     zval_ptr_stor(&obj->inner_obj);
2274:     zval_ptr_stor(&obj->inner_obj);
2275:     /* }}} */
2276:
2277: SPL_METHOD(RecursiveHashIterator, accept)
2278: {
2279:     spl_dual_iterator *intern;
2280:
2281:     if (zend_parse_parameters_none() == FAILURE) {
2282:         return;
2283:     }
2284:
2285:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2286:
2287:     if (Z_TYPE(intern->current.data) == IS_UNDEF) {
2288:         RETURN_FALSE;
2289:     } else if (Z_TYPE(intern->current.data) == IS_ARRAY) {
2290:         RETURN_BOOL(send_hash_num_elements(&INTERNAL_FUNCTION_PARAM_PASSTHRU, return_value) > 0);
2291:     }
2292:
2293:     zend_call_method_with_0_params(&obj->inner_obj, spl_recursive_hash_iterator, NULL, "accept", return_value);
2294: }
2295:
2296: #endif
2297:
2298: /* {{{ spl_dual_iterator */
2299: static void spl_dual_iterator_ctor(zend_object *object)
2300: {
2301:     spl_dual_iterator *obj = spl_dual_iterator_from_obj(object);
2302:
2303:     /* call standard ctor */
2304:     zend_object_std_init(&obj->std, &obj->std);
2305:     spl_dual_iterator_free(obj);
2306:
2307:     if (obj->inner_iterator) {
2308:         zend_iterator_ctor(obj->inner_iterator);
2309:     }
2310:
2311:     /* }}} */
2312:
2313:     /* {{{ spl_dual_iterator_free_storage */
2314:     static void spl_dual_iterator_free_storage(zend_object *object)
2315:     {
2316:         spl_dual_iterator *obj = spl_dual_iterator_from_obj(object);
2317:         spl_dual_iterator_free(obj);
2318:     }
2319:
2320:     if (IS_INSTANCE(obj->inner_obj, spl_recursive_hash_iterator)) {
2321:         zval_ptr_stor(&obj->inner_obj);
2322:     }
2323:
2324:     if (obj->dit_type == DIT_APPEND_ITERATOR) {
2325:         zend_iterator_ctor(obj->u.append_iterator);
2326:     } else if (Z_TYPE(obj->u.append.array) != IS_UNDEF) {
2327:         zval_ptr_stor(&obj->u.append.array);
2328:     }
2329:
2330:     if (obj->dit_type == DIT_CACHING_ITERATOR || obj->dit_type == DIT_RECURSIVE_CACHING_ITERATOR) {
2331:         zval_ptr_stor(&obj->u.caching_cache);
2332:     }
2333:
2334:     /* {{{ spl_dual_iterator_free */
2335:     if (obj->dit_type == DIT_CALLBACK_FILTER_ITERATOR || obj->dit_type == DIT_RECURSIVE_CALLBACK_FILTER_ITERATOR) {
2336:         if (obj->u.filter) {
2337:             spl_dual_iterator_free(obj->u.filter);
2338:         }
2339:         if (obj->u.filter_name) {
2340:             spl_dual_iterator_free(obj->u.filter_name);
2341:         }
2342:     }
2343:
2344:     /* }}} */
2345:
2346:     if (obj->dit_type == DIT_CALLBACK_FILTER_ITERATOR || obj->dit_type == DIT_RECURSIVE_CALLBACK_FILTER_ITERATOR) {
2347:         if (obj->u.filter) {
2348:             spl_dual_iterator_free(obj->u.filter);
2349:         }
2350:         if (obj->u.filter_name) {
2351:             spl_dual_iterator_free(obj->u.filter_name);
2352:         }
2353:     }
2354:
2355:     /* }}} */
2356:
2357:     zend_object_std_init(&obj->std, &obj->std);
2358:     zend_object_std_init(&obj->std, &obj->std);
2359:
2360:     /* }}} */
2361:
2362:     /* {{{ spl_dual_iterator_new */
2363:     static zend_object *spl_dual_iterator_new(zend_class_entry *class_type)
2364:     {
2365:         spl_dual_iterator *intern;
2366:
2367:         intern = zend_object_alloc(sizeof(spl_dual_iterator), class_type);
2368:         intern->dit_type = DIT_UNKNOWN;
2369:
2370:         zend_object_std_init(&intern->std, class_type);
2371:         zend_object_std_init(&intern->std, class_type);
2372:
2373:         intern->std.handlers = &spl_handlers_dual_iterator;
2374:         zend_object_std_init(&intern->std, class_type);
2375:     }
2376:     /* }}} */
2377:
2378: ZEND_BEGIN_ARG_INFO_EX(arginfo_filter_iterator_ctor, 0, 0, 1)
2379:     ZEND_ARG_INFO(0, iterator, iterator, 0)
2380: ZEND_END_ARG_INFO()
2381:
2382: static const zend_function_entry spl_filter_iterator_funcs[] = {
2383:     SPL_METHOD(FilterIterator, __construct, ZEND_ACC_PUBLIC)
2384:     SPL_METHOD(FilterIterator, rewind, ZEND_ACC_PUBLIC)
2385:     SPL_METHOD(FilterIterator, valid, ZEND_ACC_PUBLIC)
2386:     SPL_METHOD(FilterIterator, key, ZEND_ACC_PUBLIC)
2387:     SPL_METHOD(FilterIterator, current, ZEND_ACC_PUBLIC)
2388:     SPL_METHOD(FilterIterator, next, ZEND_ACC_PUBLIC)
2389:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2390:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2391:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2392:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2393:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2394:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2395:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2396:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2397:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2398:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2399:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2400:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2401:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2402:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2403:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2404:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2405:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2406:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2407:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2408:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2409:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2410:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2411:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2412:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2413:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2414:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2415:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2416:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2417:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2418:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2419:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2420:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2421:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2422:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2423:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2424:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2425:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2426:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2427:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2428:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2429:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2430:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2431:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2432:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2433:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2434:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2435:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2436:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2437:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2438:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2439:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2440:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2441:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2442:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2443:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2444:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2445:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2446:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2447:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2448:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2449:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2450:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2451:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2452:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2453:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2454:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2455:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2456:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2457:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2458:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2459:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2460:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2461:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2462:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2463:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2464:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2465:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2466:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2467:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2468:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2469:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2470:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2471:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2472:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2473:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2474:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2475:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2476:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2477:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2478:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2479:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2480:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2481:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2482:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2483:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2484:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2485:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2486:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2487:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2488:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2489:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
2490:     SPL_METHOD(FilterIterator, getIterator, ZEND_ACC_PUBLIC)
```

```

2610: ZEND_ARG_INFO(0, position);
2620: ZEND_ARG_INFO(0);
2621:
2622: static const zend_function_entry spl_funcs_limititerator[] = {
2623:     SPL_ME(LimitIterator, __construct, arginfo_limit_iterator_construct, ZEND_ACC_PUBLIC)
2624:     SPL_ME(LimitIterator, rewind, arginfo_recursew_it_void ZEND_ACC_PUBLIC)
2625:     SPL_ME(LimitIterator, valid, arginfo_recursew_it_void ZEND_ACC_PUBLIC)
2626:     SPL_ME(LimitIterator, key, arginfo_recursew_it_void ZEND_ACC_PUBLIC)
2627:     SPL_ME(LimitIterator, current, arginfo_recursew_it_void ZEND_ACC_PUBLIC)
2628:     SPL_ME(LimitIterator, next, arginfo_recursew_it_void ZEND_ACC_PUBLIC)
2629:     SPL_ME(LimitIterator, seek, arginfo_limit_iterator_seek ZEND_ACC_PUBLIC)
2630:     SPL_ME(LimitIterator, getPosition, arginfo_recursew_it_void ZEND_ACC_PUBLIC)
2631:     SPL_ME(LimitIterator, getInnerIterator, arginfo_recursew_it_void ZEND_ACC_PUBLIC)
2632:     PHP_FE_END
2633: };
2634:
2635: static inline int spl_caching_it_valid(spl_dual_it_object *intern)
2636: {
2637:     return intern->u.caching.flags & CIT_VALID ? SUCCESS : FAILURE;
2638: }
2639:
2640: static inline int spl_caching_it_has_next(spl_dual_it_object *intern)
2641: {
2642:     return spl_dual_it_valid(intern);
2643: }
2644:
2645: static inline void spl_caching_it_next(spl_dual_it_object *intern)
2646: {
2647:     if (spl_dual_it_fetch(intern, 1) == SUCCESS) {
2648:         intern->u.caching.flags |= CIT_VALID;
2649:         /* Full cache */
2650:         if (intern->u.caching.flags & CIT_FULL_CACHE) {
2651:             zval *key = &intern->current.key;
2652:             zval *data = &intern->current.data;
2653:
2654:             ZVAL_DEREF(data);
2655:             Z_TRY_ADDREF_P(data);
2656:             array_set_zval_key(Z_ARRVAL(intern->u.caching.zcache), key, data);
2657:             zval_ptr_dtor(data);
2658:         }
2659:         /* Recursion */
2660:         if (intern->vit_type == DIT_RecursiveCachingIterator) {
2661:             zval retval, szchildren, sflags;
2662:             zend_call_methodWith_O_params(&intern->inner.object, intern->inner.ce, NULL, "haschildren", &retval);
2663:             if (EG(exception)) {
2664:                 zval_ptr_dtor(&retval);
2665:                 if (intern->u.caching.flags & CIT_CATCH_GET_CHILD) {
2666:                     zend_clear_exception();
2667:                 } else {
2668:                     return;
2669:                 }
2670:             } else {
2671:                 if (zend_is_true(retval)) {
2672:                     zend_call_methodWith_O_params(&intern->inner.object, intern->inner.ce, NULL, "getchildren", &szchildren);
2673:                     if (EG(exception)) {
2674:                         zval_ptr_dtor(&szchildren);
2675:                     }
2676:                     if (intern->u.caching.flags & CIT_CATCH_GET_CHILD) {
2677:                         zend_clear_exception();
2678:                     } else {
2679:                         zval_ptr_dtor(&retval);
2680:                         return;
2681:                     }
2682:                 }
2683:                 ZVAL_LONG(&sflags, intern->u.caching.flags & CIT_PUBLIC);
2684:                 spl_instance_data_zval(&intern->u.caching.szchildren, &szchildren, &sflags);
2685:                 zval_ptr_dtor(&szchildren);
2686:             }
2687:             zval_ptr_dtor(&retval);
2688:             if (EG(exception)) {
2689:                 if (intern->u.caching.flags & CIT_CATCH_GET_CHILD) {
2690:                     zend_clear_exception();
2691:                 } else {
2692:                     return;
2693:                 }
2694:             }
2695:         }
2696:     }
2697:     if (intern->u.caching.flags & (CIT_TO_STRING_USE_INNER | CIT_CALL_TO_STRING)) {
2698:         int use_copy;
2699:         zval expr_copy;
2700:         if (intern->u.caching.flags & CIT_TO_STRING_USE_INNER) {
2701:             ZVAL_COPY_VALUE(&expr_copy, &intern->inner.object);
2702:         } else {
2703:             ZVAL_COPY_VALUE(&expr_copy, &intern->current.data);
2704:         }
2705:         use_copy = zend_make_printable_zval(&intern->u.caching.sstr, &expr_copy);
2706:         if (use_copy) {
2707:             ZVAL_COPY_VALUE(&intern->u.caching.sstr, &expr_copy);
2708:         } else {
2709:             Z_TRY_ADDREF(intern->u.caching.sstr);
2710:         }
2711:     }
2712:     spl_dual_it_next(intern, 0);
2713: } else {
2714:     intern->u.caching.flags |= ~CIT_VALID;
2715: }
2716: }
2717:
2718: static inline void spl_caching_it_rewind(spl_dual_it_object *intern)
2719: {
2720:     spl_dual_it_rewind(intern);
2721:     zend_hash_clean(Z_ARRVAL(intern->u.caching.zcache));
2722:     spl_caching_it_next(intern);
2723: }
2724:
2725: /* {{{ proto void CachingIterator::__construct (iterator $i, $flags = CIT_CALL_TO_STRING)
2726: Construct a CachingIterator from an iterator */
2727: SPL_METHOD(CachingIterator, __construct)
2728: {
2729:     spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_CachingIterator, zend_ce_iterator, DIT_CachingIterator);
2730: } /* }}} */
2731:
2732: /* {{{ proto void CachingIterator::rewind()
Rewind the iterator */
2733: SPL_METHOD(CachingIterator, rewind)
2734: {
2735:     spl_dual_it_object *intern;
2736:     spl_dual_it_object *intern;
2737:     if (zend_parse_parameters_none() == FAILURE) {
2738:         return;
2739:     }
2740:     spl_dual_it_object *intern;
2741:     if (zend_parse_parameters_none() == FAILURE) {
2742:         return;
2743:     }
2744:     spl_caching_it_rewind(intern);
2745: } /* }}} */
2746:
2747: /* {{{ proto bool CachingIterator::valid()
Check whether the current element is valid */
2748: SPL_METHOD(CachingIterator, valid)
2749: {
2750:     spl_dual_it_object *intern;
2751:     spl_dual_it_object *intern;
2752:     if (zend_parse_parameters_none() == FAILURE) {
2753:         return;
2754:     }
2755:     spl_dual_it_object *intern;
2756:     if (zend_parse_parameters_none() == FAILURE) {
2757:         return;
2758:     }
2759:     RETURN_BOOL(spl_caching_it_valid(intern) == SUCCESS);
2760: } /* }}} */
2761:
2762: /* {{{ proto void CachingIterator::next()
Move the iterator forward */
2763: SPL_METHOD(CachingIterator, next)
2764: {
2765:     spl_dual_it_object *intern;
2766:     spl_dual_it_object *intern;
2767:     if (zend_parse_parameters_none() == FAILURE) {
2768:         return;
2769:     }
2770:     spl_dual_it_object *intern;
2771:     if (zend_parse_parameters_none() == FAILURE) {
2772:         return;
2773:     }
2774:     spl_caching_it_next(intern);
2775: } /* }}} */
2776:
2777: /* {{{ proto bool CachingIterator::hasNext()
Check whether the inner iterator has a valid next element */
2778: SPL_METHOD(CachingIterator, hasNext)
2779: {
2780:     spl_dual_it_object *intern;
2781:     spl_dual_it_object *intern;
2782:     if (zend_parse_parameters_none() == FAILURE) {
2783:         return;
2784:     }
2785:     spl_dual_it_object *intern;
2786:     if (zend_parse_parameters_none() == FAILURE) {
2787:         return;
2788:     }
2789:     RETURN_BOOL(spl_caching_it_has_next(intern) == SUCCESS);
2790: } /* }}} */
2791:
2792: /* {{{ proto string CachingIterator::__toString()
Return the string representation of the current element */
2793: SPL_METHOD(CachingIterator, __toString)
2794: {
2795:     spl_dual_it_object *intern;
2796:     spl_dual_it_object *intern;
2797:     if (zend_parse_parameters_none() == FAILURE) {
2798:         return;
2799:     }
2800:     if (intern->u.caching.flags & (CIT_CALL_TO_STRING | CIT_TO_STRING_USE_KEY | CIT_TO_STRING_USE_CURRENT | CIT_TO_STRING_USE_INNER)) {
2801:         zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "has does not fetch string value (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(intern->u.caching.ce)));
2802:         return;
2803:     }
2804:     if (intern->u.caching.flags & CIT_TO_STRING_USE_KEY) {
2805:         ZVAL_COPY(&return_value, &intern->current.key);
2806:         return;
2807:     }
2808:     if (intern->u.caching.flags & CIT_CALL_TO_STRING) {
2809:         zend_throw_exception(spl_ce_InvalidArgumentException, "Missing flag CIT_TO_STRING is not possible", 0);
2810:         return;
2811:     }
2812:     if (intern->u.caching.flags & CIT_FULL_CACHE) {
2813:         zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "has does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(intern->u.caching.ce)));
2814:         return;
2815:     }
2816:     if (intern->u.caching.flags & CIT_CALL_TO_STRING) {
2817:         zend_throw_exception(spl_ce_InvalidArgumentException, "Missing flag CIT_CALL_TO_STRING is not possible", 0);
2818:         return;
2819:     }
2820:     if (intern->u.caching.flags & CIT_TO_STRING_USE_INNER) {
2821:         zend_throw_exception(spl_ce_InvalidArgumentException, "Missing flag CIT_TO_STRING_USE_INNER is not possible", 0);
2822:         return;
2823:     }
2824:     if (intern->u.caching.flags & CIT_TO_STRING_USE_KEY) {
2825:         zend_throw_exception(spl_ce_InvalidArgumentException, "Missing flag CIT_TO_STRING_USE_KEY is not possible", 0);
2826:         return;
2827:     }
2828:     if (intern->u.caching.flags & CIT_TO_STRING_USE_CURRENT) {
2829:         zend_throw_exception(spl_ce_InvalidArgumentException, "Missing flag CIT_TO_STRING_USE_CURRENT is not possible", 0);
2830:         return;
2831:     }
2832:     if (intern->u.caching.flags & CIT_TO_STRING_USE_INNER) {
2833:         zend_throw_exception(spl_ce_InvalidArgumentException, "Missing flag CIT_TO_STRING_USE_INNER is not possible", 0);
2834:         return;
2835:     }
2836:     if (intern->u.caching.flags & CIT_TO_STRING_USE_KEY) {
2837:         zend_throw_exception(spl_ce_InvalidArgumentException, "Missing flag CIT_TO_STRING_USE_KEY is not possible", 0);
2838:         return;
2839:     }
2840:     if (intern->u.caching.flags & CIT_TO_STRING_USE_CURRENT) {
2841:         zend_throw_exception(spl_ce_InvalidArgumentException, "Missing flag CIT_TO_STRING_USE_CURRENT is not possible", 0);
2842:         return;
2843:     }
2844:     if (intern->u.caching.flags & CIT_TO_STRING_USE_INNER) {
2845:         zend_throw_exception(spl_ce_InvalidArgumentException, "Missing flag CIT_TO_STRING_USE_INNER is not possible", 0);
2846:         return;
2847:     }
2848:     if (intern->u.caching.flags & CIT_TO_STRING_USE_KEY) {
2849:         zend_throw_exception(spl_ce_InvalidArgumentException, "Missing flag CIT_TO_STRING_USE_KEY is not possible", 0);
2850:         return;
2851:     }
2852:     if (intern->u.caching.flags & CIT_TO_STRING_USE_CURRENT) {
2853:         zend_throw_exception(spl_ce_InvalidArgumentException, "Missing flag CIT_TO_STRING_USE_CURRENT is not possible", 0);
2854:         return;
2855:     }
2856:     if (intern->u.caching.flags & CIT_TO_STRING_USE_INNER) {
2857:         zend_throw_exception(spl_ce_InvalidArgumentException, "Missing flag CIT_TO_STRING_USE_INNER is not possible", 0);
2858:         return;
2859:     }
2860:     if (intern->u.caching.flags & CIT_TO_STRING_USE_KEY) {
2861:         zend_throw_exception(spl_ce_InvalidArgumentException, "Missing flag CIT_TO_STRING_USE_KEY is not possible", 0);
2862:         return;
2863:     }
2864:     if (intern->u.caching.flags & CIT_TO_STRING_USE_CURRENT) {
2865:         zend_throw_exception(spl_ce_InvalidArgumentException, "Missing flag CIT_TO_STRING_USE_CURRENT is not possible", 0);
2866:         return;
2867:     }
2868:     if (intern->u.caching.flags & CIT_TO_STRING_USE_INNER) {
2869:         zend_throw
```



```

3176: // ({} proto mixed NoHwIndIterator::current())
3177: Return inner iterators current() */
3178:
3179: SPI_METHOD(NoHwIndIterator, current)
3180: {
3181:     zval *data;
3182:
3183:     if (zend_parse_parameters_none() == FAILURE) {
3184:         return;
3185:     }
3186:
3187:     SPI_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3188:     data = intern->inner.iterator->funcs->get_current_data(intern->inner.iterator);
3189:     if (data) {
3190:         ZVAL_STRING(data);
3191:     }
3192:     ZVAL_COPY(&return_value, data);
3193: }
3194: /* }}} */
3195:
3196: // ({} proto void NoHwIndIterator::next())
3197: Return inner iterators next() */
3198:
3199: SPI_METHOD(NoHwIndIterator, next)
3200: {
3201:     zval *data;
3202:
3203:     if (zend_parse_parameters_none() == FAILURE) {
3204:         return;
3205:     }
3206:
3207:     SPI_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3208:     intern->inner.iterator->funcs->move_forward(intern->inner.iterator);
3209: }
3210: /* }}} */
3211:
3212: ZEND_BEGIN_ARG_INFO(arginfo_norewind_it_construct, 0)
3213: ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
3214: ZEND_END_ARG_INFO()
3215:
3216: static const zend_function_entry spi_func_nohwinditerator[] = {
3217:     SPI_ME(NoHwIndIterator, __construct, arginfo_norewind_it_construct, ZEND_ACC_PUBLIC)
3218:     SPI_ME(NoHwIndIterator, rewind, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3219:     SPI_ME(NoHwIndIterator, valid, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3220:     SPI_ME(NoHwIndIterator, key, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3221:     SPI_ME(NoHwIndIterator, current, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3222:     SPI_ME(NoHwIndIterator, next, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3223:     SPI_ME(dual_it, getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3224:     PHP_FE_END
3225: };
3226:
3227:
3228: // ({} proto void InitInfiniteIterator::__construct(Iterator it)
3229: // Return an iterator from another iterator */
3230:
3231: SPI_METHOD(InitInfiniteIterator, __construct)
3232: {
3233:     zval *data;
3234:
3235:     if (zend_parse_parameters_none() == FAILURE) {
3236:         return;
3237:     }
3238:
3239:     SPI_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3240:
3241:     SPI_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3242:     spi_dual_it_next(intern, 1);
3243:     if (spi_dual_it_valid(intern) == SUCCESS) {
3244:         spi_dual_it_fetch(intern, 0);
3245:     } else {
3246:         spi_dual_it_rewind(intern);
3247:     }
3248:     if (spi_dual_it_valid(intern) == SUCCESS) {
3249:         spi_dual_it_fetch(intern, 0);
3250:     }
3251: }
3252: /* }}} */
3253:
3254: static const zend_function_entry spi_func_initinfiniteiterator[] = {
3255:     SPI_ME(InitInfiniteIterator, __construct, arginfo_norewind_it_construct, ZEND_ACC_PUBLIC)
3256:     SPI_ME(InitInfiniteIterator, next, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3257:     PHP_FE_END
3258: };
3259:
3260: // ({} proto void EmptyIterator::rewind())
3261: Does nothing */
3262:
3263: SPI_METHOD(EmptyIterator, rewind)
3264: {
3265:     if (zend_parse_parameters_none() == FAILURE) {
3266:         return;
3267:     }
3268: }
3269: /* }}} */
3270:
3271: // ({} proto void EmptyIterator::valid())
3272: Return false */
3273:
3274: SPI_METHOD(EmptyIterator, valid)
3275: {
3276:     if (zend_parse_parameters_none() == FAILURE) {
3277:         return;
3278:     }
3279:
3280:     return FALSE;
3281: }
3282: /* }}} */
3283:
3284: // ({} proto void EmptyIterator::key())
3285: Throws exception BadMethodCallException */
3286:
3287: SPI_METHOD(EmptyIterator, key)
3288: {
3289:     if (zend_parse_parameters_none() == FAILURE) {
3290:         return;
3291:     }
3292:
3293:     send_throw_exception(spi_ce_BadMethodCallException, "Accessing the key of an EmptyIterator", 0);
3294: }
3295: /* }}} */
3296:
3297: // ({} proto void EmptyIterator::current())
3298: Throws exception BadMethodCallException */
3299:
3300: SPI_METHOD(EmptyIterator, current)
3301: {
3302:     if (zend_parse_parameters_none() == FAILURE) {
3303:         return;
3304:     }
3305:
3306:     send_throw_exception(spi_ce_BadMethodCallException, "Accessing the value of an EmptyIterator", 0);
3307: }
3308: /* }}} */
3309:
3310: // ({} proto void EmptyIterator::next())
3311: Does nothing */
3312:
3313: SPI_METHOD(EmptyIterator, next)
3314: {
3315:     if (zend_parse_parameters_none() == FAILURE) {
3316:         return;
3317:     }
3318: }
3319: /* }}} */
3320:
3321: static const zend_function_entry spi_func_emptyiterator[] = {
3322:     SPI_ME(EmptyIterator, rewind, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3323:     SPI_ME(EmptyIterator, valid, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3324:     SPI_ME(EmptyIterator, key, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3325:     SPI_ME(EmptyIterator, current, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3326:     SPI_ME(EmptyIterator, next, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3327:     PHP_FE_END
3328: };
3329:
3330:
3331: int spi_append_it_next_iterator(spi_dual_it_object *intern) /* {{{ */
3332: {
3333:     if (!IS_EMPTY(intern->inner.sobjiect)) {
3334:         zval_ptr_dtor(&intern->inner.sobjiect);
3335:         ZVAL_EMPTY_OBJ(intern->inner.sobjiect);
3336:         intern->inner.ce = NULL;
3337:         if (intern->inner.iterator) {
3338:             zend_iterator_dtor(intern->inner.iterator);
3339:             intern->inner.iterator = NULL;
3340:         }
3341:     }
3342:
3343:     if (intern->v.append.iterator->funcs->valid(intern->v.append.iterator) == SUCCESS) {
3344:         zval *it;
3345:
3346:         it = intern->v.append.iterator->funcs->get_current_data(intern->v.append.iterator);
3347:         ZVAL_COPY(&intern->inner.sobjiect, it);
3348:         intern->inner.ce = Z_OBJECT_P(1);
3349:         intern->inner.iterator = intern->inner.ce->get_iterator(intern->inner.ce, it, 0);
3350:         spi_dual_it_rewind(intern);
3351:         return SUCCESS;
3352:     } else {
3353:         return FAILURE;
3354:     }
3355: }
3356: /* }}} */
3357:
3358: void spi_append_it_fetch(spi_dual_it_object *intern) /* {{{ */
3359: {
3360:     while (spi_dual_it_valid(intern) != SUCCESS) {
3361:         intern->v.append.iterator->funcs->move_forward(intern->v.append.iterator);
3362:         if (spi_append_it_next_iterator(intern) != SUCCESS) {
3363:             return;
3364:         }
3365:     }
3366:
3367:     spi_dual_it_fetch(intern, 0);
3368: }
3369: /* }}} */
3370:
3371: void spi_append_it_next(spi_dual_it_object *intern) /* {{{ */
3372: {
3373:     if (spi_dual_it_valid(intern) == SUCCESS) {
3374:         spi_dual_it_next(intern, 1);
3375:     }
3376:
3377:     spi_append_it_fetch(intern);
3378: }
3379: /* }}} */
3380:
3381: // ({} proto void AppendIterator::__construct()

```

```

3364: SPL_METHOD (AppendIterator, __construct,
3365: {
3366:     spl_dual_it_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_AppendIterator, zend_ce_iterator, DIT_AppendIterator);
3367: } /* }}} */
3368:
3369: /* {{{ proto void AppendIterator::append(Iterator $t)
3370:  * Append an Iterator */
3371: SPL_METHOD (AppendIterator, append)
3372: {
3373:     spl_dual_it_object *intern;
3374:     zval *it;
3375:
3376:     SPL_FETCH_AND_CHECK_DUAL_IT (intern, getThis());
3377:
3378:     if (zend_parse_parameters_ex (ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "O", &it, zend_ce_iterator) == FAILURE) {
3379:         return;
3380:     }
3381:     if (intern->u.append_iterator->funcs->valid (intern->u.append_iterator) == SUCCESS && spl_dual_it_valid (intern) != SUCCESS) {
3382:         spl_array_iterator_append (intern->u.append_array, it);
3383:         intern->u.append_iterator->funcs->move_forward (intern->u.append_iterator);
3384:     } else {
3385:         spl_array_iterator_append (intern->u.append_array, it);
3386:     }
3387:
3388:     if (intern->inner_iterator != spl_dual_it_valid (intern)) != SUCCESS {
3389:         if (intern->u.append_iterator->funcs->valid (intern->u.append_iterator) != SUCCESS) {
3390:             intern->u.append_iterator->funcs->rewind (intern->u.append_iterator);
3391:         }
3392:         do {
3393:             spl_append_it_next_iterator (intern);
3394:         } while (!IS_TRUE (intern->inner_subj) != Z_OBJ_P (it));
3395:         spl_append_it_fetch (intern);
3396:     }
3397: } /* }}} */
3398:
3399: /* {{{ proto mixed AppendIterator::current()
3400:  * Get the current element value */
3401: SPL_METHOD (AppendIterator, current)
3402: {
3403:     spl_dual_it_object *intern;
3404:
3405:     if (zend_parse_parameters_none() == FAILURE) {
3406:         return;
3407:     }
3408:
3409:     SPL_FETCH_AND_CHECK_DUAL_IT (intern, getThis());
3410:
3411:     spl_dual_it_fetch (intern, 1);
3412:     if (Z_TYPE (intern->current_data) != IS_UNDEF) {
3413:         zval *value = &intern->current_data;
3414:
3415:         ZVAL_DEREF (value);
3416:         ZVAL_COPY (return_value, value);
3417:     } else {
3418:         RETURN_NULL();
3419:     }
3420: } /* }}} */
3421:
3422: /* {{{ proto void AppendIterator::rewind()
3423:  * Rewind to the first iterator and rewind the first iterator, too */
3424: SPL_METHOD (AppendIterator, rewind)
3425: {
3426:     spl_dual_it_object *intern;
3427:
3428:     if (zend_parse_parameters_none() == FAILURE) {
3429:         return;
3430:     }
3431:
3432:     SPL_FETCH_AND_CHECK_DUAL_IT (intern, getThis());
3433:
3434:     intern->u.append_iterator->funcs->rewind (intern->u.append_iterator);
3435:     if (spl_append_it_next_iterator (intern) == SUCCESS) {
3436:         spl_append_it_fetch (intern);
3437:     }
3438: } /* }}} */
3439:
3440: /* {{{ proto bool AppendIterator::valid()
3441:  * Check if the current state is valid */
3442: SPL_METHOD (AppendIterator, valid)
3443: {
3444:     spl_dual_it_object *intern;
3445:
3446:     if (zend_parse_parameters_none() == FAILURE) {
3447:         return;
3448:     }
3449:
3450:     SPL_FETCH_AND_CHECK_DUAL_IT (intern, getThis());
3451:
3452:     RETURN_BOOL (Z_TYPE (intern->current_data) != IS_UNDEF);
3453: } /* }}} */
3454:
3455: /* {{{ proto void AppendIterator::next()
3456:  * Forward to next element */
3457: SPL_METHOD (AppendIterator, next)
3458: {
3459:     spl_dual_it_object *intern;
3460:
3461:     if (zend_parse_parameters_none() == FAILURE) {
3462:         return;
3463:     }
3464:
3465:     SPL_FETCH_AND_CHECK_DUAL_IT (intern, getThis());
3466:
3467:     spl_append_it_next (intern);
3468: } /* }}} */
3469:
3470: /* {{{ proto int AppendIterator::getIteratorIndex()
3471:  * Get index of iterator */
3472: SPL_METHOD (AppendIterator, getIteratorIndex)
3473: {
3474:     spl_dual_it_object *intern;
3475:
3476:     if (zend_parse_parameters_none() == FAILURE) {
3477:         return;
3478:     }
3479:
3480:     SPL_FETCH_AND_CHECK_DUAL_IT (intern, getThis());
3481:
3482:     APPEND_IT_CHECK_CTOR (intern);
3483:     spl_array_iterator_key (&intern->u.append_array, return_value);
3484: } /* }}} */
3485:
3486: /* {{{ proto ArrayIterator AppendIterator::getArrayIterator()
3487:  * Get access to inner ArrayIterator */
3488: SPL_METHOD (AppendIterator, getArrayIterator)
3489: {
3490:     spl_dual_it_object *intern;
3491:     zval *value;
3492:
3493:     if (zend_parse_parameters_none() == FAILURE) {
3494:         return;
3495:     }
3496:
3497:     SPL_FETCH_AND_CHECK_DUAL_IT (intern, getThis());
3498:
3499:     value = &intern->u.append_array;
3500:     ZVAL_DEREF (value);
3501:     ZVAL_COPY (return_value, value);
3502: } /* }}} */
3503:
3504: ZEND_BEGIN_ARG_INFO (arginfo_append_it_append, 0)
3505:     ZEND_ARG_OBJ_INFO (0, iterator, Iterator, 0)
3506: ZEND_END_ARG_INFO ();
3507:
3508: static const zend_function_entry spl_funcs_AppendIterator[] = {
3509:     SPL_ME (AppendIterator, __construct, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3510:     SPL_ME (AppendIterator, append, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3511:     SPL_ME (AppendIterator, rewind, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3512:     SPL_ME (AppendIterator, valid, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3513:     SPL_ME (Dual_it, key, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3514:     SPL_ME (AppendIterator, current, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3515:     SPL_ME (AppendIterator, next, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3516:     SPL_ME (Dual_it, getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3517:     SPL_ME (AppendIterator, getIteratorIndex, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3518:     SPL_ME (AppendIterator, getArrayIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3519:     PHP_FE_END
3520: };
3521:
3522: PHP_API int spl_iterator_apply (zval *obj, spl_iterator_apply_func_t apply_func, void *puser)
3523: {
3524:     zend_object_iterator *iter;
3525:     zend_class_entry *ce = Z_OBJCE_P (obj);
3526:
3527:     iter = ce->get_iterator (ce, obj, 0);
3528:
3529:     if (EG (exception)) {
3530:         goto done;
3531:     }
3532:
3533:     iter->index = 0;
3534:     if (iter->funcs->rewind() {
3535:         iter->funcs->rewind (iter);
3536:         if (EG (exception)) {
3537:             goto done;
3538:         }
3539:     }
3540:
3541:     while (iter->funcs->valid (iter) == SUCCESS) {
3542:         if (EG (exception)) {
3543:             goto done;
3544:         }
3545:         if (apply_func (iter, puser) == ZEND_HASH_APPLY_STOP || EG (exception)) {
3546:             goto done;
3547:         }
3548:         iter->index++;
3549:         iter->funcs->move_forward (iter);
3550:         if (EG (exception)) {
3551:             goto done;
3552:         }
3553:     }
3554:
3555:     static int spl_iterator_to_array_apply (zend_object_iterator *iter, void *puser) /* {{{ */
3556:     {
3557:         zval *data, *return_value = (zval *) puser;
3558:
3559:         data = iter->funcs->get_current_data (iter);
3560:         if (EG (exception)) {
3561:             return ZEND_HASH_APPLY_STOP;
3562:         }
3563:         if (data == NULL) {
3564:             return ZEND_HASH_APPLY_STOP;
3565:         }
3566:         if (iter->funcs->get_current_key () {
3567:             zval key;
3568:             iter->funcs->get_current_key (iter, &key);
3569:             if (EG (exception)) {
3570:                 return ZEND_HASH_APPLY_STOP;
3571:             }
3572:             array_set_zval_key (&return_value, (return_value), &key, data);
3573:             zval_ptr_dtor (&key);
3574:             else {
3575:                 Z_TRY_ADDREF_P (data);
3576:                 add_next_index_zval (return_value, data);
3577:             }
3578:             return ZEND_HASH_APPLY_KEEP;
3579:         }
3580:     } /* }}} */
3581:
3582: static int spl_iterator_to_values_apply (zend_object_iterator *iter, void *puser) /* {{{ */
3583:     {
3584:         zval *data, *return_value = (zval *) puser;
3585:
3586:         data = iter->funcs->get_current_data (iter);
3587:         if (EG (exception)) {
3588:             return ZEND_HASH_APPLY_STOP;
3589:         }
3590:         if (data == NULL) {
3591:             return ZEND_HASH_APPLY_STOP;
3592:         }
3593:         Z_TRY_ADDREF_P (data);
3594:         add_next_index_zval (return_value, data);
3595:         return ZEND_HASH_APPLY_KEEP;
3596:     } /* }}} */
3597:
3598: /* {{{ proto array iterator_to_array(Traversable $t, bool use_keys = true)
3599:  * Copy the iterator into an array */
3600: PHP_FUNCTION (iterator_to_array)
3601: {
3602:     zval *obj;
3603:     zend_bool use_keys = 1;
3604:
3605:     if (zend_parse_parameters (ZEND_NUM_ARGS(), "O|b", &obj, zend_ce_traversable, &use_keys) == FAILURE) {
3606:         RETURN_FALSE;
3607:     }
3608:     array_init (&return_value);
3609:
3610:     if (spl_iterator_apply (obj, use_keys ? spl_iterator_to_array_apply : spl_iterator_to_values_apply, (void *) return_value) != SUCCESS) {
3611:         spl_ptr_dtor (&return_value);
3612:         RETURN_NULL();
3613:     }
3614: } /* }}} */
3615:
3616: static int spl_iterator_count_apply (zend_object_iterator *iter, void *puser) /* {{{ */
3617:     {
3618:         if (zend_long *puser)++;
3619:         return ZEND_HASH_APPLY_KEEP;
3620:     } /* }}} */
3621:
3622: /* {{{ proto int iterator_count(Traversable $t)
3623:  * Count the elements in an iterator */
3624: PHP_FUNCTION (iterator_count)
3625: {
3626:     zval *obj;
3627:     zend_long count = 0;
3628:
3629:     if (zend_parse_parameters (ZEND_NUM_ARGS(), "O", &obj, zend_ce_traversable) == FAILURE) {
3630:         RETURN_FALSE;
3631:     }
3632:
3633:     if (spl_iterator_apply (obj, spl_iterator_count_apply, (void *) &count) == SUCCESS) {
3634:         RETURN_LONG (count);
3635:     }
3636: } /* }}} */
3637:
3638: typedef struct {
3639:     zval *obj;
3640:     zval *return_value;
3641:     zend_long count;
3642:     zend_fcall_info fci;
3643:     zend_fcall_info_cache fcc;
3644:     spl_iterator_apply_info;
3645: }
3646:
364
```

```
3738: REGISTER_SPL_STD_CLASS_EX(IteratorIterator, spl_dual_it_new, spl_funcns_IteratorIterator);
3739: REGISTER_SPL_ITERATOR(IteratorIterator);
3740: REGISTER_SPL_ITERATOR(IteratorIterator);
3741: REGISTER_SPL_IMPLMENTS(IteratorIterator, OuterIterator);
3742:
3743: REGISTER_SPL_SUB_CLASS_EX(FilterIterator, IteratorIterator, spl_dual_it_new, spl_funcns_FilterIterator);
3744: spl_on_FilterIterator->on_flags |= ZEND_ACC_EXPLICIT_ABSTRACT_CLASS;
3745:
3746: REGISTER_SPL_SUB_CLASS_EX(RecursiveFilterIterator, FilterIterator, spl_dual_it_new, spl_funcns_RecursiveFilterIterator);
3747: REGISTER_SPL_IMPLMENTS(RecursiveFilterIterator, RecursiveIterator);
3748:
3749: REGISTER_SPL_SUB_CLASS_EX(CallbackFilterIterator, FilterIterator, spl_dual_it_new, spl_funcns_CallbackFilterIterator);
3750:
3751: REGISTER_SPL_SUB_CLASS_EX(RecursiveCallbackFilterIterator, CallbackFilterIterator, spl_dual_it_new, spl_funcns_RecursiveCallbackFilterIterator);
3752: REGISTER_SPL_IMPLMENTS(RecursiveCallbackFilterIterator, RecursiveIterator);
3753:
3754:
3755: REGISTER_SPL_SUB_CLASS_EX(ParentIterator, RecursiveFilterIterator, spl_dual_it_new, spl_funcns_ParentIterator);
3756:
3757: REGISTER_SPL_INTERFACE(SeekableIterator);
3758: REGISTER_SPL_ITERATOR(SeekableIterator);
3759:
3760: REGISTER_SPL_SUB_CLASS_EX(LimitIterator, IteratorIterator, spl_dual_it_new, spl_funcns_LimitIterator);
3761:
3762: REGISTER_SPL_SUB_CLASS_EX(CachingIterator, IteratorIterator, spl_dual_it_new, spl_funcns_CachingIterator);
3763: REGISTER_SPL_IMPLMENTS(CachingIterator, ArrayAccess);
3764: REGISTER_SPL_IMPLMENTS(CachingIterator, Countable);
3765:
3766: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "CALL_TOSTRING", CIT_CALL_TOSTRING);
3767: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "CATCH_GET_CHILD", CIT_CATCH_GET_CHILD);
3768: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "TOSTRING_USE_KEY", CIT_TOSTRING_USE_KEY);
3769: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "TOSTRING_USE_CURRENT", CIT_TOSTRING_USE_CURRENT);
3770: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "TOSTRING_USE_INNER", CIT_TOSTRING_USE_INNER);
3771: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "FULL_CACHE", CIT_FULL_CACHE);
3772:
3773: REGISTER_SPL_SUB_CLASS_EX(RecursiveCachingIterator, CachingIterator, spl_dual_it_new, spl_funcns_RecursiveCachingIterator);
3774: REGISTER_SPL_IMPLMENTS(RecursiveCachingIterator, RecursiveIterator);
3775:
3776: REGISTER_SPL_SUB_CLASS_EX(NoRewindIterator, IteratorIterator, spl_dual_it_new, spl_funcns_NoRewindIterator);
3777:
3778: REGISTER_SPL_SUB_CLASS_EX(AppendIterator, IteratorIterator, spl_dual_it_new, spl_funcns_AppendIterator);
3779:
3780: REGISTER_SPL_IMPLMENTS(RecursiveIteratorIterator, OuterIterator);
3781:
3782: REGISTER_SPL_SUB_CLASS_EX(InfinitelIterator, IteratorIterator, spl_dual_it_new, spl_funcns_InfinitelIterator);
3783:
3784: #if HAVE_PCRE || HAVE_REGEX_PCRE
3785: REGISTER_SPL_SUB_CLASS_EX(RegexIterator, FilterIterator, spl_dual_it_new, spl_funcns_RegexIterator);
3786: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "USE_KEY", REGIT_USE_KEY);
3787: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "INVERT_MATCH", REGIT_INVERTED);
3788: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "MATCH", REGIT_MATCH_MATCH);
3789: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "GET_MATCH", REGIT_MATCH_GET_MATCH);
3790: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "ALL_MATCHES", REGIT_MATCH_ALL_MATCHES);
3791: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "SPLIT", REGIT_MATCH_SPLIT);
3792: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "REPLACE", REGIT_MATCH_REPLACE);
3793: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "PROPERTY", REGIT_MATCH_PROPERTY);
3794: REGISTER_SPL_IMPLMENTS(RecursiveRegexIterator, RecursiveIterator);
3795:
3796: spl_on_RegexIterator = NULL;
3797: spl_on_RecursiveRegexIterator = NULL;
3798: #endif
3799:
3800: REGISTER_SPL_STD_CLASS_EX(EmptyIterator, NULL, spl_funcns_EmptyIterator);
3801: REGISTER_SPL_ITERATOR(EmptyIterator);
3802:
3803: REGISTER_SPL_SUB_CLASS_EX(RecursiveTreeIterator, RecursiveIteratorIterator, spl_RecursiveTreeIterator_new, spl_funcns_RecursiveTreeIterator);
3804: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "BYPASS_CURRENT", RTIT_BYPASS_CURRENT);
3805: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "BYPASS_KEY", RTIT_BYPASS_KEY);
3806: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_LEFT", 0);
3807: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_MID_HAS_NEXT", 1);
3808: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_MID_LAST", 2);
3809: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_END_HAS_NEXT", 3);
3810: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_END_LAST", 4);
3811: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_RIGHT", 5);
3812:
3813: return SUCCESS;
3814: }
3815: /* }}} */
3816:
3817: /*
3818:  * Local variables:
3819:  * tab-width: 4
3820:  * c-basic-offset: 4
3821:  * End:
3822:  * vim600: fdm=marker
3823:  * vim: noet sw=4 ts=4
3824:  */
```

```
1: /*
2:  *-----*
3:  * | PHP Version 7 |
4:  *-----*
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  *-----*
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  *-----*
15:  * | Authors: Etienne Kneuss <colder@php.net> |
16:  *-----*
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_DLIST_H
22: #define SPL_DLIST_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26:
27: extern PHPAPI zend_class_entry *spl_ce_SplDoublyLinkedList;
28: extern PHPAPI zend_class_entry *spl_ce_SplQueue;
29: extern PHPAPI zend_class_entry *spl_ce_SplStack;
30:
31: PHP_MINIT_FUNCTION(spl_dlist);
32:
33: #endif /* SPL_DLIST_H */
34:
35: /*
36:  * Local Variables:
37:  * n-basis-offset: 4
38:  * tab-width: 4
39:  * End:
40:  * vim600: fdm=marker
41:  * vim: noet sw=4 ts=4
42:  */
```

```
1: /*
2:  * =====
3:  * | PHP Version ? |
4:  * =====
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * =====
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * =====
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * =====
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef PHP_FUNCTIONS_H
22: #define PHP_FUNCTIONS_H
23:
24: #include "php.h"
25:
26: typedef zend_object* (*create_object_func_t)(zend_class_entry *class_type);
27:
28: #define REGISTER_SPL_STO_CLASS(class_name, obj_ctor) \
29:     spl_register_std_class(spl_ce_ ## class_name, # class_name, obj_ctor, NULL);
30:
31: #define REGISTER_SPL_STO_CLASS_EX(class_name, obj_ctor, funcns) \
32:     spl_register_std_class(spl_ce_ ## class_name, # class_name, obj_ctor, funcns);
33:
34: #define REGISTER_SPL_SUB_CLASS_EX(class_name, parent_class_name, obj_ctor, funcns) \
35:     spl_register_sub_class(spl_ce_ ## class_name, spl_ce_ ## parent_class_name, # class_name, obj_ctor, funcns);
36:
37: #define REGISTER_SPL_INTERFACE(class_name) \
38:     spl_register_interface(spl_ce_ ## class_name, # class_name, spl_funcns_ ## class_name);
39:
40: #define REGISTER_SPL_IMPLEMENTNS(class_name, interface_name) \
41:     zend_class_implements(spl_ce_ ## class_name, 1, spl_ce_ ## interface_name);
42:
43: #define REGISTER_SPL_ITERATOR(class_name) \
44:     zend_class_implements(spl_ce_ ## class_name, 1, zend_ce_iterator);
45:
46: #define REGISTER_SPL_PROPERTY(class_name, prop_name, prop_flags) \
47:     spl_register_property(spl_ce_ ## class_name, prop_name, sizeof(prop_name)-1, prop_flags);
48:
49: #define REGISTER_SPL_CLASS_CONST_LONG(class_name, const_name, value) \
50:     zend_declare_class_constant_long(spl_ce_ ## class_name, const_name, sizeof(const_name)-1, (zend_long)value);
51:
52: void spl_register_std_class(zend_class_entry ** pcew, char * class_name, create_object_func_t ctor, const zend_function_entry * function_list);
53: void spl_register_sub_class(zend_class_entry ** pcew, zend_class_entry * parent_ce, char * class_name, create_object_func_t ctor, const zend_function_
ntry * function_list);
54: void spl_register_interface(zend_class_entry ** pcew, char * class_name, const zend_function_entry * functions);
55:
56: void spl_register_property(zend_class_entry * class_entry, char *prop_name, int prop_name_len, int prop_flags);
57:
58: /* sub: whether to allow subclasses/interfaces
59:    allow = 0: allow all classes and interfaces
60:    allow > 0: allow all that match and mask ce_flags
61:    allow < 0: disallow all that match and mask ce_flags
62: */
63: void spl_add_class_name(zval * list, zend_class_entry * pce, int allow, int ce_flags);
64: void spl_add_interfaces(zval * list, zend_class_entry * pce, int allow, int ce_flags);
65: void spl_add_traits(zval * list, zend_class_entry * pce, int allow, int ce_flags);
66: int spl_add_classes(zend_class_entry *pce, zval *list, int sub, int allow, int ce_flags);
67:
68: /* caller must free(return) */
69: zend_string *spl_get_private_prop_name(zend_class_entry *ce, char *prop_name, int prop_len);
70:
71: #define SPL_ME(class_name, function_name, arg_info, flags) \
72:     PHP_ME(spl, ## class_name, function_name, arg_info, flags)
73:
74: #define SPL_ABSTRACT_ME(class_name, function_name, arg_info) \
75:     ZEND_ABSTRACT_ME(spl, ## class_name, function_name, arg_info)
76:
77: #define SPL_METHOD(class_name, function_name) \
78:     PHP_METHOD(spl, ## class_name, function_name)
79:
80: #define SPL_MA(class_name, function_name, alias_class, alias_function, arg_info, flags) \
81:     PHP_ALIAS(spl, ## alias_class, function_name, alias_function, arg_info, flags)
82: #endif /* PHP_FUNCTIONS_H */
83:
84: /*
85:  * Local Variables:
86:  * c-basic-offset: 4
87:  * tab-width: 4
88:  * End:
89:  * vim600: fdm=marker
90:  * vim: noet sw=4 ts=4
91:  */
```

[illegible]

```

377:  if (alif->func_ptr &&
378:      UNEXPECTED(alif->func_ptr->common.fn_flags & ZEND_ACC_CALL_VIA_TRAMPOLINE)) {
379:      zend_string_release(alif->func_ptr->common.function_name);
380:      zend_free_trampoline(alif->func_ptr);
381:  }
382:  if (!IS_UNDEF(alif->closure)) {
383:      zval_ptr_dtor(alif->closure);
384:  }
385:  efree(alif);
386:  }
387:
388:  /* {{{ proto void spl_autoload_call(string class_name)
389:   * Try all registered autoload function to load the requested class */
390:  PHP_FUNCTION(spl_autoload_call)
391:  {
392:      zval *class_name, *retval;
393:      zend_string *lc_name, *func_name;
394:      autoload_func_info *alif;
395:  }
396:  if (zend_parse_parameters(ZEND_NUM_ARGS() | "s", &class_name) == FAILURE || Z_TYPE_P(class_name) != IS_STRING) {
397:      return;
398:  }
399:  }
400:  if (SP1_G(autoload_functions)) {
401:      zend_position pos;
402:      zend_ulong num_idx;
403:      zend_function *func;
404:      zend_fcall_info fci;
405:      zend_fcall_info_cache fci_cache;
406:      zend_class_entry *called_scope = zend_get_called_scope(execute_data);
407:      int i_automload_running = SP1_G(autoload_running);
408:  }
409:  SP1_G(autoload_running) = 1;
410:  lc_name = zend_string_tolower(Z_STR_P(class_name));
411:  }
412:  fci.size = sizeof(fci);
413:  fci.retval = &retval;
414:  fci.param_count = 1;
415:  fci.params = class_name;
416:  fci.no_separation = 1;
417:  }
418:  ZVAL_UNDEF(&fci.function_name); /* Unused */
419:  }
420:  zend_hash_internal_pointer_reset_ex(SP1_G(autoload_functions), &pos);
421:  while (zend_hash_get_current_key_ex(SP1_G(autoload_functions), &func_name, &num_idx, &pos) == HASH_KEY_IS_STRING) {
422:      alif = zend_hash_get_current_data_ptr_ex(SP1_G(autoload_functions), &pos);
423:      func = alif->func_ptr;
424:      if (UNEXPECTED(func->common.fn_flags & ZEND_ACC_CALL_VIA_TRAMPOLINE)) {
425:          func = &alif->closure;
426:          memcpy(func, alif->func_ptr, sizeof(send_up_array));
427:          zend_string_addr(func->op_array.function_name);
428:      }
429:      ZVAL_UNDEF(&ret_val);
430:      fci.func_handler = func;
431:      if (!IS_UNDEF(alif->obj)) {
432:          fci.object = NULL;
433:          fci.object = NULL;
434:          fci.calling_scope = alif->obj;
435:          if (alif->obj &&
436:              (called_scope ||
437:               !instanceof_function(called_scope, alif->obj))) {
438:              fci.called_scope = alif->obj;
439:          } else {
440:              fci.called_scope = called_scope;
441:          }
442:          fci.obj = fci.obj->obj;
443:          fci.obj = fci.obj->obj;
444:          fci.called_scope = Z_OBJCE(alif->obj);
445:          fci.called_scope = Z_OBJCE(alif->obj);
446:      }
447:  }
448:  zend_call_function(&fci, &fci_cache);
449:  zval_ptr_dtor(&ret_val);
450:  }
451:  if (EG(exception)) {
452:      break;
453:  }
454:  if (pos + 1 == SP1_G(autoload_functions)->nNumOfElements) {
455:      zend_hash_exists(&EG(class_table), lc_name);
456:      break;
457:  }
458:  zend_hash_move_forward_ex(SP1_G(autoload_functions), &pos);
459:  }
460:  zend_string_release(lc_name);
461:  SP1_G(autoload_running) = 1;
462:  } else {
463:      /* do not use or overwrite EG(autoload_func) here */
464:      zend_call_method_with_1_params(NULL, NULL, NULL, "spl_autoload", NULL, class_name);
465:  }
466:  }
467:  /* }}} */
468:  }
469:  zend_time HT_MOVE_TAIL_TO_HEAD(ht)
470:  {
471:      Bucket tmp = (ht->data[(ht->nNumUsed-1)]);
472:      memmove(&ht->data + 1, &ht->data, \
473:          sizeof(Bucket) * (ht->nNumUsed - 1));
474:      (ht->data[0]) = tmp;
475:      zend_hash_rehash(ht);
476:      } while (0)
477:  }
478:  /* {{{ proto bool spl_autoload_register([mixed autoload_function [, bool throw [, bool prepend]])
479:   * Register given function as __autoload() implementation */
480:  PHP_FUNCTION(spl_autoload_register)
481:  {
482:      zend_string *func_name;
483:      char *error = NULL;
484:      zend_string *lc_name;
485:      zval *callable = NULL;
486:      zend_bool do_throw = 1;
487:      zend_bool prepend = 0;
488:      zend_function *spl_func_ptr;
489:      autoload_func_info alif;
490:      zend_object *obj_ptr;
491:      zend_fcall_info_cache fci;
492:  }
493:  if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS() | "sb", &callable, &do_throw, &prepend) == FAILURE) {
494:      return;
495:  }
496:  if (ZEND_NUM_ARGS() > 0) {
497:      if (!is_callable_ex(callable, NULL, IS_CALLABLE_STRICT, &func_name, &fci, &error)) {
498:          alif.obj = fci.calling_scope;
499:          alif.func_ptr = fci.function_handler;
500:          obj_ptr = fci.object;
501:          if (Z_TYPE_P(callable) == IS_ARRAY) {
502:              if (obj_ptr && alif.func_ptr && !alif.func_ptr->common.fn_flags & ZEND_ACC_STATIC) {
503:                  if (do_throw) {
504:                      zend_throw_exception_ex(spl_ce_LogicException, 0, "Passed array specifies a non static method but no object (ts)", error);
505:                  }
506:              }
507:              if (error) {
508:                  efree(error);
509:              }
510:              zend_string_release(func_name);
511:              RETURN_FALSE;
512:          } else if (do_throw) {
513:              zend_throw_exception_ex(spl_ce_LogicException, 0, "Passed array does not specify to method (ts)", alif.func_ptr ? "a callable" : "an exist
ng", obj_ptr ? "static" : "", error);
514:          }
515:          if (error) {
516:              efree(error);
517:          }
518:          zend_string_release(func_name);
519:          RETURN_FALSE;
520:      } else if (!Z_TYPE_P(callable) == IS_STRING) {
521:          if (do_throw) {
522:              zend_throw_exception_ex(spl_ce_LogicException, 0, "Function 'ts' not to (ts)", ZSTR_VAL(func_name), alif.func_ptr ? "callable" : "found", err
or);
523:          }
524:          if (error) {
525:              efree(error);
526:          }
527:          zend_string_release(func_name);
528:          RETURN_FALSE;
529:      } else {
530:          if (do_throw) {
531:              zend_throw_exception_ex(spl_ce_LogicException, 0, "Illegal value passed (ts)", error);
532:          }
533:          if (error) {
534:              efree(error);
535:          }
536:          zend_string_release(func_name);
537:          RETURN_FALSE;
538:      }
539:      if (fci.func_handler->type == ZEND_INTERNAL_FUNCTION &&
540:          fci.func_handler->internal_function_handler == &f_spl_autoload_call) {
541:          if (do_throw) {
542:              zend_throw_exception_ex(spl_ce_LogicException, 0, "Function spl_autoload_call() cannot be registered");
543:          }
544:          if (error) {
545:              efree(error);
546:          }
547:          zend_string_release(func_name);
548:          RETURN_FALSE;
549:      }
550:      alif.obj = fci.calling_scope;
551:      alif.func_ptr = fci.function_handler;
552:      obj_ptr = fci.object;
553:      if (error) {
554:          efree(error);
555:      }
556:  }
557:  if (Z_TYPE_P(callable) == IS_OBJECT) {
558:      ZVAL_COPY(&alif.closure, callable);
559:  }
560:  lc_name = zend_string_alloc(ZSTR_LEN(func_name) + sizeof(uint32_t), 0);
561:  zend_str_tolower_copy(ZSTR_VAL(lc_name), ZSTR_VAL(func_name));
562:  memcpy(ZSTR_VAL(lc_name) + ZSTR_LEN(func_name), &Z_OBJ_HANDLE_P(callable), sizeof(uint32_t));

```

```
751:     return;
752: }
753: RETURN_FALSE;
754: }
755:
756: fptr = zend_hash_str_find_ptr(EG(function_table), "spl_autoload_call", sizeof("spl_autoload_call") - 1);
757:
758: if (EG(autoload_func) == fptr) {
759:     zend_string *key;
760:     array_init(&return_value);
761:     ZEND_HASH_FOREACH_STR_KEY_PTR(SPL_G(autoload_functions), key, aif) {
762:         if (!ZENDREF(aif->closure)) {
763:             Z_ADDREF(aif->closure);
764:             add_next_index_val(&return_value, &aif->closure);
765:         } else if (aif->func_ptr->common.scope) {
766:             zval tmp;
767:
768:             array_init(&tmp);
769:             if (!ZENDREF(aif->obj)) {
770:                 Z_ADDREF(aif->obj);
771:                 add_next_index_val(&tmp, &aif->obj);
772:             } else {
773:                 add_next_index_str(&tmp, zend_string_copy(aif->ce->name));
774:             }
775:             add_next_index_str(&tmp, zend_string_copy(aif->func_ptr->common.function_name));
776:             add_next_index_val(&return_value, &tmp);
777:         } else {
778:             if (ZENDREF(EG(VAL(aif->func_ptr->common.function_name), "_lambda_func", sizeof("_lambda_func") - 1)) {
779:                 add_next_index_str(&return_value, zend_string_copy(aif->func_ptr->common.function_name));
780:             } else {
781:                 add_next_index_str(&return_value, zend_string_copy(key));
782:             }
783:         }
784:     } ZEND_HASH_FOREACH_END();
785:     return;
786: }
787:
788: array_init(&return_value);
789: add_next_index_str(&return_value, zend_string_copy(EG(autoload_func)->common.function_name));
790: /* }}} */
791:
792: /* {{{ proto string spl_object_hash(object obj)
793:  * Returns hash id for given object */
794: PHP_FUNCTION(spl_object_hash)
795: {
796:     zval *obj;
797:
798:     if (zend_parse_parameters(ZEND_NUM_ARGS() & "o", &obj) == FAILURE) {
799:         return;
800:     }
801:
802:     RETURN_NEW_STR(spl_php_object_hash(obj));
803: }
804: /* }}} */
805:
806: /* {{{ proto int spl_object_id(object obj)
807:  * Returns the integer object handle for the given object */
808: PHP_FUNCTION(spl_object_id)
809: {
810:     zval *obj;
811:
812:     ZEND_PARSE_PARAMETERS_START(1, 1)
813:     Z_PARAM_OBJECT(obj)
814:     ZEND_PARSE_PARAMETERS_END();
815:
816:     RETURN_LONG((zend_long)Z_OBJ_HANDLE_P(obj));
817: }
818: /* }}} */
819:
820: PHPAPI zend_string *spl_php_object_hash(zval *obj) /* {{{ */
821: {
822:     intptr_t hash_handle, hash_handlers;
823:
824:     if (!SPL_G(hash_mask_init)) {
825:         SPL_G(hash_mask_handle) = (intptr_t)(php_mt_rand() >> 1);
826:         SPL_G(hash_mask_handlers) = (intptr_t)(php_mt_rand() >> 1);
827:         SPL_G(hash_mask_init) = 1;
828:     }
829:
830:     hash_handle = SPL_G(hash_mask_handle) * (intptr_t)Z_OBJ_HANDLE_P(obj);
831:     hash_handlers = SPL_G(hash_mask_handlers);
832:
833:     return strprintf(32, "%06x%06x", hash_handle, hash_handlers);
834: }
835: /* }}} */
836:
837: int spl_build_class_list_string(zval *entry, char **list) /* {{{ */
838: {
839:     char *res;
840:
841:     sprintf(res, 0, "%s", *list, Z_STWAL_P(entry));
842:     zfree(*list);
843:     *list = res;
844:     return ZEND_HASH_APPLY_KEEP;
845: } /* }}} */
846:
847: /* {{{ PHP_MININFO(spl)
848: */
849: PHP_MININFO_FUNCTION(spl)
850: {
851:     zval list;
852:     char *str;
853:
854:     php_info_print_table_start();
855:     php_info_print_table_header(2, "SPL support", "enabled");
856:
857:     array_init(&list);
858:     SPL_LIST_CLASSES(&list, 0, 1, ZEND_ACC_INTERFACE);
859:     str = estrdup("");
860:     zend_hash_apply_with_argument(Z_ARRVAL_P(&list), (apply_func_arg_t)spl_build_class_list_string, &str);
861:     zval_dtor(&list);
862:     php_info_print_table_row(2, "Interfaces", str + 2);
863:     zfree(str);
864:
865:     array_init(&list);
866:     SPL_LIST_CLASSES(&list, 0, -1, ZEND_ACC_INTERFACE);
867:     str = estrdup("");
868:     zend_hash_apply_with_argument(Z_ARRVAL_P(&list), (apply_func_arg_t)spl_build_class_list_string, &str);
869:     zval_dtor(&list);
870:     php_info_print_table_row(2, "Classes", str + 2);
871:     zfree(str);
872:
873:     php_info_print_table_end();
874: }
875: /* }}} */
876:
877: /* {{{ arginfo */
878: ZEND_BEGIN_ARG_INFO_EX(arginfo_iterator_to_array, 0, 0, 1)
879:     ZEND_ARG_CBT_INFO(0, iterator, Traversable, 0)
880:     ZEND_ARG_INFO(0, use_keys)
881: ZEND_END_ARG_INFO();
882:
883: ZEND_BEGIN_ARG_INFO_EX(arginfo_iterator, 0)
884:     ZEND_ARG_CBT_INFO(0, iterator, Traversable, 0)
885: ZEND_END_ARG_INFO();
886:
887: ZEND_BEGIN_ARG_INFO_EX(arginfo_iterator_apply, 0, 0, 2)
888:     ZEND_ARG_CBT_INFO(0, iterator, Traversable, 0)
889:     ZEND_ARG_INFO(0, function)
890:     ZEND_ARG_ARRAY_INFO(0, args, 1)
891: ZEND_END_ARG_INFO();
892:
893: ZEND_BEGIN_ARG_INFO_EX(arginfo_class_parents, 0, 0, 1)
894:     ZEND_ARG_INFO(0, instance)
895:     ZEND_ARG_INFO(0, autoload)
896: ZEND_END_ARG_INFO();
897:
898: ZEND_BEGIN_ARG_INFO_EX(arginfo_class_implements, 0, 0, 1)
899:     ZEND_ARG_INFO(0, what)
900:     ZEND_ARG_INFO(0, autoload)
901: ZEND_END_ARG_INFO();
902:
903: ZEND_BEGIN_ARG_INFO_EX(arginfo_class_uses, 0, 0, 1)
904:     ZEND_ARG_INFO(0, what)
905:     ZEND_ARG_INFO(0, autoload)
906: ZEND_END_ARG_INFO();
907:
908:
909: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_classes, 0)
910:     ZEND_ARG_INFO(0)
911:
912: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_functions, 0)
913:     ZEND_ARG_INFO(0)
914:
915: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload, 0, 0, 1)
916:     ZEND_ARG_INFO(0, class_name)
917:     ZEND_ARG_INFO(0, file_extensions)
918: ZEND_END_ARG_INFO();
919:
920: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_extensions, 0, 0, 0)
921:     ZEND_ARG_INFO(0, file_extensions)
922: ZEND_END_ARG_INFO();
923:
924: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_call, 0, 0, 1)
925:     ZEND_ARG_INFO(0, class_name)
926: ZEND_END_ARG_INFO();
927:
928: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_register, 0, 0, 0)
929:     ZEND_ARG_INFO(0, autoload_function)
930:     ZEND_ARG_INFO(0, throw)
931:     ZEND_ARG_INFO(0, prepend)
932: ZEND_END_ARG_INFO();
933:
934: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_unregister, 0, 0, 1)
935:     ZEND_ARG_INFO(0, autoload_function)
936: ZEND_END_ARG_INFO();
937:
938: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_object_hash, 0, 0, 1)
```

```
939:     ZEND_ARG_INFO(0, obj)
940: ZEND_END_ARG_INFO();
941:
942: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_object_id, 0, 0, 1)
943:     ZEND_ARG_INFO(0, obj)
944: ZEND_END_ARG_INFO();
945: /* }}} */
946:
947: /* {{{ spl_functions
948: */
949: static const zend_function_entry spl_functions[] = {
950:     PHP_FE(spl_classes, arginfo_spl_classes)
951:     PHP_FE(spl_autoload, arginfo_spl_autoload)
952:     PHP_FE(spl_autoload_extensions, arginfo_spl_autoload_extensions)
953:     PHP_FE(spl_autoload_register, arginfo_spl_autoload_register)
954:     PHP_FE(spl_autoload_unregister, arginfo_spl_autoload_unregister)
955:     PHP_FE(spl_autoload_functions, arginfo_spl_autoload_functions)
956:     PHP_FE(spl_autoload_call, arginfo_spl_autoload_call)
957:     PHP_FE(class_parents, arginfo_class_parents)
958:     PHP_FE(class_implements, arginfo_class_implements)
959:     PHP_FE(class_uses, arginfo_class_uses)
960:     PHP_FE(spl_object_hash, arginfo_spl_object_hash)
961:     PHP_FE(spl_object_id, arginfo_spl_object_id)
962: #ifdef SPL_ITERATORS_H
963:     PHP_FE(iterator_to_array, arginfo_iterator_to_array)
964:     PHP_FE(iterator_count, arginfo_iterator)
965:     PHP_FE(iterator_apply, arginfo_iterator_apply)
966: #endif /* SPL_ITERATORS_H */
967:     PHP_FE_END
968: };
969: /* }}} */
970:
971: /* {{{ PHP_MINIT_FUNCTION(spl)
972: */
973: PHP_MINIT_FUNCTION(spl)
974: {
975:     PHP_MINIT(spl_exceptions) (INIT_FUNC_ARGS_PASSTHRU);
976:     PHP_MINIT(spl_iterators) (INIT_FUNC_ARGS_PASSTHRU);
977:     PHP_MINIT(spl_array) (INIT_FUNC_ARGS_PASSTHRU);
978:     PHP_MINIT(spl_directory) (INIT_FUNC_ARGS_PASSTHRU);
979:     PHP_MINIT(spl_dlist) (INIT_FUNC_ARGS_PASSTHRU);
980:     PHP_MINIT(spl_heap) (INIT_FUNC_ARGS_PASSTHRU);
981:     PHP_MINIT(spl_directory) (INIT_FUNC_ARGS_PASSTHRU);
982:     PHP_MINIT(spl_observer) (INIT_FUNC_ARGS_PASSTHRU);
983:
984:     return SUCCESS;
985: }
986: /* }}} */
987:
988: PHP_RINIT_FUNCTION(spl) /* {{{ */
989: {
990:     SPL_G(autoload_extensions) = NULL;
991:     SPL_G(autoload_functions) = NULL;
992:     SPL_G(hash_mask_init) = 0;
993:     return SUCCESS;
994: } /* }}} */
995:
996: PHP_SHUTDOWN_FUNCTION(spl) /* {{{ */
997: {
998:     if (SPL_G(autoload_extensions)) {
999:         zend_string_release(SPL_G(autoload_extensions));
1000:         SPL_G(autoload_extensions) = NULL;
1001:     }
1002:     if (SPL_G(autoload_functions)) {
1003:         zend_hash_destroy(SPL_G(autoload_functions));
1004:         FREE_HASHTABLE(SPL_G(autoload_functions));
1005:         SPL_G(autoload_functions) = NULL;
1006:     }
1007:     if (SPL_G(hash_mask_init)) {
1008:         SPL_G(hash_mask_init) = 0;
1009:     }
1010:     return SUCCESS;
1011: } /* }}} */
1012:
1013: /* {{{ spl_module_entry
1014: */
1015: zend_module_entry spl_module_entry = {
1016:     STANDARD_MODULE_HEADER,
1017:     "spl",
1018:     spl_functions,
1019:     PHP_MINIT(spl),
1020:     NULL,
1021:     PHP_RINIT(spl),
1022:     PHP_SHUTDOWN(spl),
1023:     PHP_MININFO(spl),
1024:     PHP_SPL_VERSION,
1025:     PHP_MODULE_GLOBALS(spl),
1026:     PHP_GINIT(spl),
1027:     NULL,
1028:     NULL,
1029:     STANDARD_MODULE_PROPERTIES_EX
1030: };
1031: /* }}} */
1032:
1033: /*
1034:  * Local variables:
1035:  * tab-width: 4
1036:  * c-basic-offset: 4
1037:  * End:
1038:  * vim600: fdm=marker
1039:  * vim: noet sw=4 ts=4
1040:  */
```



```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifdef HAVE_CONFIG_H
22: #include "config.h"
23: #endif
24:
25: #include "php.h"
26: #include "php_ini.h"
27: #include "ext/standard/info.h"
28: #include "zend_interfaces.h"
29: #include "zend_exceptions.h"
30:
31: #include "spl_spl.h"
32: #include "spl_functions.h"
33: #include "spl_engine.h"
34: #include "spl_exceptions.h"
35:
36: PHPAPI zend_class_entry *spl_ce_LogicException;
37: PHPAPI zend_class_entry *spl_ce_BadFunctionCallException;
38: PHPAPI zend_class_entry *spl_ce_BadMethodCallException;
39: PHPAPI zend_class_entry *spl_ce_DomainException;
40: PHPAPI zend_class_entry *spl_ce_InvalidArgumentException;
41: PHPAPI zend_class_entry *spl_ce_LengthException;
42: PHPAPI zend_class_entry *spl_ce_OutOfRangeException;
43: PHPAPI zend_class_entry *spl_ce_RuntimeException;
44: PHPAPI zend_class_entry *spl_ce_OutOfBoundsException;
45: PHPAPI zend_class_entry *spl_ce_OverflowException;
46: PHPAPI zend_class_entry *spl_ce_RangeException;
47: PHPAPI zend_class_entry *spl_ce_UnderflowException;
48: PHPAPI zend_class_entry *spl_ce_UnexpectedValueException;
49:
50: #define spl_ce_Exception zend_ce_exception
51:
52: /* {{{ PHP_MINIT_FUNCTION(spl_exceptions) */
53: PHP_MINIT_FUNCTION(spl_exceptions)
54: {
55:     REGISTER_SPL_SUB_CLASS_EX(LogicException, Exception, NULL, NULL);
56:     REGISTER_SPL_SUB_CLASS_EX(BadFunctionCallException, LogicException, NULL, NULL);
57:     REGISTER_SPL_SUB_CLASS_EX(BadMethodCallException, BadFunctionCallException, NULL, NULL);
58:     REGISTER_SPL_SUB_CLASS_EX(DomainException, LogicException, NULL, NULL);
59:     REGISTER_SPL_SUB_CLASS_EX(InvalidArgumentException, LogicException, NULL, NULL);
60:     REGISTER_SPL_SUB_CLASS_EX(LengthException, LogicException, NULL, NULL);
61:     REGISTER_SPL_SUB_CLASS_EX(OutOfRangeException, LogicException, NULL, NULL);
62:
63:     REGISTER_SPL_SUB_CLASS_EX(RuntimeException, Exception, NULL, NULL);
64:     REGISTER_SPL_SUB_CLASS_EX(OutOfBoundsException, RuntimeException, NULL, NULL);
65:     REGISTER_SPL_SUB_CLASS_EX(OverflowException, RuntimeException, NULL, NULL);
66:     REGISTER_SPL_SUB_CLASS_EX(RangeException, RuntimeException, NULL, NULL);
67:     REGISTER_SPL_SUB_CLASS_EX(UnderflowException, RuntimeException, NULL, NULL);
68:     REGISTER_SPL_SUB_CLASS_EX(UnexpectedValueException, RuntimeException, NULL, NULL);
69:
70:     return SUCCESS;
71: }
72: /* }}} */
73:
74: /*
75:  * Local variables:
76:  * tab-width: 4
77:  * c-basic-offset: 4
78:  * End:
79:  * vim600: fdm=marker
80:  * vim: noet sw=4 ts=4
81:  */
```

```
1: /*
2:  *-----
3:  * | PHP Version 7 |
4:  *-----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  *-----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  *-----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  *-----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_OBSERVER_H
22: #define SPL_OBSERVER_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26:
27: extern PHPAPI zend_class_entry *spl_ce_SplObserver;
28: extern PHPAPI zend_class_entry *spl_ce_SplSubject;
29: extern PHPAPI zend_class_entry *spl_ce_SplObjectStorage;
30: extern PHPAPI zend_class_entry *spl_ce_MultiPlaiterator;
31:
32: PHP_MINIT_FUNCTION(spl_observer);
33:
34: #endif /* SPL_OBSERVER_H */
35:
36: /*
37:  * Local Variables:
38:  * c-basic-offset: 4
39:  * tab-width: 4
40:  * End:
41:  * vim600: fdm=marker
42:  * vim: noet sw=4 ts=4
43:  */
```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_EXCEPTIONS_H
22: #define SPL_EXCEPTIONS_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26:
27: extern PHPAPI zend_class_entry *spl_ce_LogicalException;
28: extern PHPAPI zend_class_entry *spl_ce_BadFunctionCallException;
29: extern PHPAPI zend_class_entry *spl_ce_BadMethodCallException;
30: extern PHPAPI zend_class_entry *spl_ce_DomainException;
31: extern PHPAPI zend_class_entry *spl_ce_InvalidArgumentException;
32: extern PHPAPI zend_class_entry *spl_ce_LengthException;
33: extern PHPAPI zend_class_entry *spl_ce_OutOfRangeException;
34:
35: extern PHPAPI zend_class_entry *spl_ce_RuntimeException;
36: extern PHPAPI zend_class_entry *spl_ce_OutOfBoundsException;
37: extern PHPAPI zend_class_entry *spl_ce_OverflowException;
38: extern PHPAPI zend_class_entry *spl_ce_RangeException;
39: extern PHPAPI zend_class_entry *spl_ce_UnderflowException;
40: extern PHPAPI zend_class_entry *spl_ce_UnexpectedValueException;
41:
42: PHP_MINIT_FUNCTION(spl_exceptions);
43:
44: #endif /* SPL_EXCEPTIONS_H */
45:
46: /*
47:  * Local Variables:
48:  * c-basic-offset: 4
49:  * tab-width: 4
50:  * End:
51:  * vim600: fdm=marker
52:  * vim: noet sw=4 ts=4
53:  */
```

```
1: /*
2:  * =====
3:  * | PHP Version 7 |
4:  * =====
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * =====
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * =====
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * =====
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef HAVE_CONFIG_H
22: #include "config.h"
23: #endif
24:
25: #include "php.h"
26: #include "php_ini.h"
27: #include "ext/standard/info.h"
28: #include "php_spl.h"
29:
30: /* {{{ spl_register_interface */
31: void spl_register_interface(zend_class_entry ** pce, char * class_name, const zend_function_entry * functions)
32: {
33:     zend_class_entry ce;
34:
35:     INIT_CLASS_ENTRY_EX(ce, class_name, strlen(class_name), functions);
36:     *pce = zend_register_internal_interface(&ce);
37: }
38: /* }}} */
39:
40: /* {{{ spl_register_std_class */
41: PHPAPI void spl_register_std_class(zend_class_entry ** pce, char * class_name, void * obj_ctor, const zend_function_entry * function_list)
42: {
43:     zend_class_entry ce;
44:
45:     INIT_CLASS_ENTRY_EX(ce, class_name, strlen(class_name), function_list);
46:     *pce = zend_register_internal_class(&ce);
47:
48:     /* entries changed by initialize */
49:     if (obj_ctor) {
50:         (*pce->create_object) = obj_ctor;
51:     }
52: }
53: /* }}} */
54:
55: /* {{{ spl_register_sub_class */
56: PHPAPI void spl_register_sub_class(zend_class_entry ** pce, zend_class_entry * parent_ce, char * class_name, void * obj_ctor, const zend_function_entry * function_list)
57: {
58:     zend_class_entry ce;
59:
60:     INIT_CLASS_ENTRY_EX(ce, class_name, strlen(class_name), function_list);
61:     *pce = zend_register_internal_class_ex(&ce, parent_ce);
62:
63:     /* entries changed by initialize */
64:     if (obj_ctor) {
65:         (*pce->create_object) = obj_ctor;
66:     } else {
67:         (*pce->create_object) = parent_ce->create_object;
68:     }
69: }
70: /* }}} */
71:
72: /* {{{ spl_register_property */
73: void spl_register_property(zend_class_entry * class_entry, char *prop_name, int prop_name_len, int prop_flags)
74: {
75:     zend_declare_property_null(class_entry, prop_name, prop_name_len, prop_flags);
76: }
77: /* }}} */
78:
79: /* {{{ spl_add_class_name */
80: void spl_add_class_name(zval *list, zend_class_entry *pce, int allow, int ce_flags)
81: {
82:     if ((allow && (0 < pce->ce_flags & ce_flags)) || (allow && ! (pce->ce_flags & ce_flags))) {
83:         zval *tmp;
84:
85:         if ((tmp = zend_hash_find(Z_ARRVAL_P(list), pce->name)) == NULL) {
86:             zval tmp;
87:             ZVAL_STR_COPY(&tmp, pce->name);
88:             zend_hash_add(Z_ARRVAL_P(list), pce->name, &tmp);
89:         }
90:     }
91: }
92: /* }}} */
93:
94: /* {{{ spl_add_interfaces */
95: void spl_add_interfaces(zval *list, zend_class_entry * pce, int allow, int ce_flags)
96: {
97:     uint32_t num_interfaces;
98:
99:     for (num_interfaces = 0; num_interfaces < pce->num_interfaces; num_interfaces++) {
100:         spl_add_class_name(list, pce->interfaces[num_interfaces], allow, ce_flags);
101:     }
102: }
103: /* }}} */
104:
105: /* {{{ spl_add_traits */
106: void spl_add_traits(zval *list, zend_class_entry * pce, int allow, int ce_flags)
107: {
108:     uint32_t num_traits;
109:
110:     for (num_traits = 0; num_traits < pce->num_traits; num_traits++) {
111:         spl_add_class_name(list, pce->traits[num_traits], allow, ce_flags);
112:     }
113: }
114: /* }}} */
115:
116: /* {{{ spl_add_classes */
117: int spl_add_classes(zend_class_entry *pce, zval *list, int sub, int allow, int ce_flags)
118: {
119:     if (!pce) {
120:         return 0;
121:     }
122:     spl_add_class_name(list, pce, allow, ce_flags);
123:     if (sub) {
124:         spl_add_interfaces(list, pce, allow, ce_flags);
125:         while (pce->parent) {
126:             pce = pce->parent;
127:             spl_add_classes(pce, list, sub, allow, ce_flags);
128:         }
129:     }
130:     return 0;
131: }
132: /* }}} */
133:
134: zend_string * spl_get_private_prop_name(zend_class_entry *ce, char *prop_name, int prop_len) /* {{{ */
135: {
136:     return zend_mangle_property_name(ZSTR_VAL(ce->name), ZSTR_LEN(ce->name), prop_name, prop_len, 0);
137: }
138: /* }}} */
139:
140: /*
141:  * Local Variables:
142:  * tab-width: 4
143:  * c-basic-offset: 4
144:  * End:
145:  * vim600: fdm=marker
146:  * vim: noet sw=4 ts=4
147:  */
148: */
```

```
1: /*
2:  *
3:  * PHP Version 7
4:  *
5:  * Copyright (c) 1997-2018 The PHP Group
6:  *
7:  * This source file is subject to version 3.01 of the PHP license,
8:  * that is bundled with this package in the file LICENSE, and is
9:  * available through the world-wide-web at the following url:
10:  * https://www.php.net/license/3.01.txt
11:  * If you did not receive a copy of the PHP license and are unable to
12:  * obtain it through the world-wide-web, please send a note to
13:  * license@php.net so we can mail you a copy immediately.
14:  *
15:  * Authors: Marcus Boerger <helly@php.net>
16:  *
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_DIRECTORY_H
22: #define SPL_DIRECTORY_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26:
27: extern PHPAPI zend_class_entry *spl_ce_SplFileInfo;
28: extern PHPAPI zend_class_entry *spl_ce_DirectoryIterator;
29: extern PHPAPI zend_class_entry *spl_ce_FilesystemIterator;
30: extern PHPAPI zend_class_entry *spl_ce_RecursiveDirectoryIterator;
31: extern PHPAPI zend_class_entry *spl_ce_GlobIterator;
32: extern PHPAPI zend_class_entry *spl_ce_SplFileInfo;
33: extern PHPAPI zend_class_entry *spl_ce_SplTempFileObject;
34:
35: PHP_MINIT_FUNCTION(spl_directory);
36:
37: typedef enum {
38:     SPL_FS_INFO, /* must be 0 */
39:     SPL_FS_DIR,
40:     SPL_FS_FILE,
41: } SPL_FS_OBJ_TYPE;
42:
43: typedef struct _spl_filesystem_object spl_filesystem_object;
44:
45: typedef void (*spl_foreign_dtor_t)(spl_filesystem_object *object);
46: typedef void (*spl_foreign_clone_t)(spl_filesystem_object *src, spl_filesystem_object *dst);
47:
48: PHPAPI char* spl_filesystem_object_get_path(spl_filesystem_object *intern, size_t *len);
49:
50: typedef struct _spl_other_handler {
51:     spl_foreign_dtor_t dtor;
52:     spl_foreign_clone_t clone;
53: } spl_other_handler;
54:
55: /* define an overloaded iterator structure */
56: typedef struct {
57:     zend_object_iterator intern;
58:     zval *current;
59:     void *object;
60: } spl_filesystem_iterator;
61:
62: struct _spl_filesystem_object {
63:     void *other;
64:     const spl_other_handler *other_handler;
65:     char *_path;
66:     size_t _path_len;
67:     char *_full_path;
68:     char *_file_name;
69:     size_t file_name_len;
70:     SPL_FS_OBJ_TYPE type;
71:     zend_long flags;
72:     zend_class_entry *file_class;
73:     zend_class_entry *info_class;
74:     union {
75:         struct {
76:             php_stream *stream;
77:             php_stream_dirent *entry;
78:             char *sub_path;
79:             size_t sub_path_len;
80:             int index;
81:             int is_recursive;
82:             zend_function *func_read;
83:             zend_function *func_next;
84:             zend_function *func_valid;
85:         } dir;
86:         struct {
87:             php_stream *stream;
88:             php_stream_context *context;
89:             zval *scontext;
90:             char *open_mode;
91:             size_t open_mode_len;
92:             zval current_zval;
93:             char *current_line;
94:             size_t current_line_len;
95:             size_t max_line_len;
96:             zend_long current_line_num;
97:             zval zresource;
98:             zend_function *func_getCurr;
99:             char *delimater;
100:             char *enclosure;
101:             char *escape;
102:         } file;
103:     } u;
104:     zend_object std;
105: };
106:
107: static inline spl_filesystem_object *spl_filesystem_from_obj(zend_object *obj) /* {{{ */ {
108:     return (spl_filesystem_object*)((char*)(obj) - XOffsetOf(spl_filesystem_object, std));
109: }
110: /* }}} */
111:
112: #define Z_SPL_FILESYSTEM_P(zv) spl_filesystem_from_obj(Z_OBJ_P(zv))
113:
114: static inline spl_filesystem_iterator *spl_filesystem_object_to_iterator(spl_filesystem_object *obj)
115: {
116:     spl_filesystem_iterator *it;
117:
118:     it = ecalloc(1, sizeof(spl_filesystem_iterator));
119:     it->object = (void *)obj;
120:     zend_iterator_init(it->intern);
121:     return it;
122: }
123:
124: static inline spl_filesystem_object* spl_filesystem_iterator_to_object(spl_filesystem_iterator *it)
125: {
126:     return (spl_filesystem_object*)it->object;
127: }
128:
129: #define SPL_FILE_OBJECT_DROP_NEW_LINE 0x00000001 /* drop new lines */
130: #define SPL_FILE_OBJECT_READ_HEADER 0x00000002 /* read or read/next */
131: #define SPL_FILE_OBJECT_SKIP_EMPTY 0x00000004 /* skip empty lines */
132: #define SPL_FILE_OBJECT_READ_CSV 0x00000008 /* read via fgets */
133: #define SPL_FILE_OBJECT_MASK 0x0000000F /* read via fgets */
134:
135: #define SPL_FILE_DIR_CURRENT_AS_FILEINFO 0x00000000 /* make RecursiveDirectoryTree::current() return SplFileInfo */
136: #define SPL_FILE_DIR_CURRENT_AS_ZIP 0x00000001 /* make RecursiveDirectoryTree::current() return getZipInfo() */
137: #define SPL_FILE_DIR_CURRENT_AS_PATHNAME 0x00000020 /* make RecursiveDirectoryTree::current() return getPathname() */
138: #define SPL_FILE_DIR_CURRENT_MODE_MASK 0x000000F0 /* mask RecursiveDirectoryTree::current() */
139: #define SPL_FILE_DIR_CURRENT_MODE_MASK ((intern->flags & SPL_FILE_DIR_CURRENT_MODE_MASK) ~ mode)
140:
141: #define SPL_FILE_DIR_KEY_AS_PATHNAME 0x00000000 /* make RecursiveDirectoryTree::key() return getPathname() */
142: #define SPL_FILE_DIR_KEY_AS_PATHNAME 0x00000000 /* make RecursiveDirectoryTree::key() return getZipInfo() */
143: #define SPL_FILE_DIR_FOLLOW_SYMLINKS 0x00000020 /* make RecursiveDirectoryTree::hasChildren() follow symlinks */
144: #define SPL_FILE_DIR_KEY_MODE_MASK 0x000000F0 /* mask RecursiveDirectoryTree::key() */
145: #define SPL_FILE_DIR_KEY_MODE_MASK ((intern->flags & SPL_FILE_DIR_KEY_MODE_MASK) ~ mode)
146:
147: #define SPL_FILE_DIR_SKIPDOTS 0x00000100 /* Tells whether it should skip dots or not */
148: #define SPL_FILE_DIR_UNIFYPATHS 0x00000200 /* whether to unify path separators */
149: #define SPL_FILE_DIR_OTHERS_MASK 0x00000300 /* mask used for get/setFlags */
150:
151: #endif /* SPL_DIRECTORY_H */
152:
153: /*
154:  * Local Variables:
155:  * c-basic-offset: 4
156:  * tab-width: 4
157:  * End:
158:  * vim600: fdm=marker
159:  * vim: noet sw=4 ts=4
160:  */
```

```
1: /*
2:  * PHP Version 7
3:  *
4:  * Copyright (c) 1997-2018 The PHP Group
5:  *
6:  * This source file is subject to version 3.01 of the PHP license,
7:  * that is bundled with this package in the file LICENSE, and is
8:  * available through the world-wide-web at the following url:
9:  * http://www.php.net/license/3_01.txt
10:  * If you did not receive a copy of the PHP license and are unable to
11:  * obtain it through the world-wide-web, please send a note to
12:  * license@php.net so we can mail you a copy immediately.
13:  *
14:  * Author: Antony Duvall <tony@daylesday.org>
15:  *
16:  * Elienne Kneuss <elienne@php.net>
17:  */
18:
19:
20: /* $Id$ */
21:
22: #ifndef HAVE_CONFIG_H
23: #include "config.h"
24: #endif
25:
26: #include "php.h"
27: #include "php_ini.h"
28: #include "ext/standard/info.h"
29: #include "zend_exceptions.h"
30:
31: #include "spl_spl.h"
32: #include "spl_functions.h"
33: #include "spl_engine.h"
34: #include "spl_fixedarray.h"
35: #include "spl_exceptions.h"
36: #include "spl_iterators.h"
37:
38: zend_object_handlers spl_handler_SplFixedArray;
39: PHP_API zend_class_entry *spl_ce_SplFixedArray;
40:
41: #ifndef COMPILE_DL_SPL_FIXEDARRAY
42: #error COMPILE_DL_SPL_FIXEDARRAY
43: #endif
44:
45: typedef struct _spl_fixedarray { /* {{{ */
46:     zend_long size;
47:     zval *elements;
48: } spl_fixedarray;
49: /* }}} */
50:
51: typedef struct _spl_fixedarray_object { /* {{{ */
52:     spl_fixedarray array;
53:     zend_function *fptr_offset_get;
54:     zend_function *fptr_offset_set;
55:     zend_function *fptr_offset_has;
56:     zend_function *fptr_offset_del;
57:     zend_function *fptr_count;
58:     int current;
59:     int flags;
60:     zend_class_entry *ce_iterator;
61:     zend_object std;
62: } spl_fixedarray_object;
63: /* }}} */
64:
65: typedef struct _spl_fixedarray_it { /* {{{ */
66:     zend_user_iterator intern;
67: } spl_fixedarray_it;
68: /* }}} */
69:
70: #define SPL_FIXEDARRAY_OVERLOADED_REWIND 0x0001
71: #define SPL_FIXEDARRAY_OVERLOADED_VALID 0x0002
72: #define SPL_FIXEDARRAY_OVERLOADED_KEY 0x0004
73: #define SPL_FIXEDARRAY_OVERLOADED_CURRENT 0x0008
74: #define SPL_FIXEDARRAY_OVERLOADED_NEXT 0x0010
75:
76: static inline spl_fixedarray_object *spl_fixed_array_from_obj(zend_object *obj) /* {{{ */ {
77:     return (spl_fixedarray_object*)((char*)obj) - XFFOFFSET(spl_fixedarray_object, std);
78: }
79: /* }}} */
80:
81: #define SPL_FIXEDARRAY_P(rv) spl_fixed_array_from_obj(Z_OBJ_P(rv))
82:
83: static void spl_fixedarray_init(spl_fixedarray *array, zend_long size) /* {{{ */ {
84:     if (size > 0) {
85:         array->size = 0; /* reset size in case realloc() fails */
86:         array->elements = ecalloc(size, sizeof(zval));
87:         array->size = size;
88:     } else {
89:         array->elements = NULL;
90:         array->size = 0;
91:     }
92: }
93: /* }}} */
94:
95: static void spl_fixedarray_resize(spl_fixedarray *array, zend_long size) /* {{{ */ {
96:     if (size == array->size) {
97:         /* nothing to do */
98:         return;
99:     }
100:
101:     /* first initialization */
102:     if (array->size == 0) {
103:         spl_fixedarray_init(array, size);
104:         return;
105:     }
106:
107:     /* clearing the array */
108:     if (size == 0) {
109:         zend_long i;
110:
111:         for (i = 0; i < array->size; i++) {
112:             zval_ptr_dtor(&(array->elements[i]));
113:         }
114:
115:         if (array->elements) {
116:             zfree(array->elements);
117:             array->elements = NULL;
118:         }
119:     } else if (size > array->size) {
120:         array->elements = safe_realloc(array->elements, size, sizeof(zval), 0);
121:         memset(array->elements + array->size, '\0', sizeof(zval) * (size - array->size));
122:     } else { /* size < array->size */
123:         zend_long i;
124:
125:         for (i = size; i < array->size; i++) {
126:             zval_ptr_dtor(&(array->elements[i]));
127:         }
128:
129:         array->elements = erealloc(array->elements, sizeof(zval) * size);
130:     }
131:
132:     array->size = size;
133: }
134: /* }}} */
135:
136: static void spl_fixedarray_copy(spl_fixedarray *to, spl_fixedarray *from) /* {{{ */ {
137:     zend_long i;
138:
139:     for (i = 0; i < from->size; i++) {
140:         ZVAL_COPY(&(to->elements[i]), &(from->elements[i]));
141:     }
142: }
143: /* }}} */
144:
145: static HashTable* spl_fixedarray_object_get_ge(zval *obj, zval **table, int *n) /* {{{ */ {
146:     spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(obj);
147:     HashTable *ht = zend_std_get_properties(obj);
148:
149:     *table = intern->array.elements;
150:     *n = (int)intern->array.size;
151:
152:     return ht;
153: }
154: /* }}} */
155:
156: static HashTable* spl_fixedarray_object_get_properties(zval *obj) /* {{{ */ {
157:     spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(obj);
158:     HashTable *ht = zend_std_get_properties(obj);
159:     zend_long i = 0;
160:
161:     if (intern->array.size > 0) {
162:         zend_long j = zend_hash_num_elements(ht);
163:
164:         for (i = 0; i < intern->array.size; i++) {
165:             if (IS_UNDEF(intern->array.elements[i])) {
166:                 zend_hash_index_update(ht, i, spl_uninitialized_zval);
167:             }
168:             Z_STR_ASSIGN(&(intern->array.elements[i]), spl_uninitialized_zval);
169:         }
170:     } else {
171:         zend_hash_index_update(ht, 1, spl_uninitialized_zval);
172:     }
173:
174:     if (j > intern->array.size) {
175:         for (i = intern->array.size; i < j; i++) {
176:             zend_hash_index_del(ht, i);
177:         }
178:     }
179:
180:     return ht;
181: }
182: /* }}} */
183:
184: static void spl_fixedarray_object_free_storage(zend_object *object) /* {{{ */ {
185:     spl_fixedarray_object *intern = spl_fixed_array_from_obj(object);
186:
187:     zend_long i;
188:
189:     for (i = 0; i < intern->array.size; i++) {
190:         zval_ptr_dtor(&(intern->array.elements[i]));
191:     }
192:
193:     if (intern->array.elements) {
194:         zfree(intern->array.elements);
195:     }
196:
197:     zend_object_std_dtor((zend_object_std*)object);
198: }
199: /* }}} */
200:
201: static zend_object *spl_fixedarray_object_new(zend_class_entry *ce, zval *obj, int by_ref) /* {{{ */ {
202:     spl_fixedarray_object *object;
203:     zend_class_entry *parent = ce->parent;
204:     int inherited = 0;
205:
206:     object = zend_object_alloc(sizeof(spl_fixedarray_object), parent);
207:     zend_object_std_init((zend_object_std*)object, ce);
208:     object->properties_table = zend_hash_init(ce->properties_table, 0, NULL, NULL, 0);
209:     intern = (spl_fixedarray_object*)object;
210:     intern->current = 0;
211:     intern->flags = 0;
212:
213:     if (obj && !ce->clone_obj) {
214:         spl_fixedarray_object *other = Z_SPLFIXEDARRAY_P(obj);
215:         intern->ce_get_iterator = other->ce_get_iterator;
216:         spl_fixedarray_init(intern->array, other->array.size);
217:         spl_fixedarray_copy(intern->array, other->array);
218:     }
219:
220:     while (parent) {
221:         if (parent == spl_ce_SplFixedArray) {
222:             intern->std.handlers = spl_handler_SplFixedArray;
223:             class_entry->get_iterator = spl_fixedarray_get_iterator;
224:             break;
225:         }
226:         parent = parent->parent;
227:         inherited = 1;
228:     }
229:
230:     if (!parent) { /* this must never happen */
231:         php_error_fatal("Internal compiler error, Class is not child of SplFixedArray");
232:     }
233:
234:     if (class_entry->iterator_funcs.if_rewind == zend_hash_str_find_ptr(class_entry->function_table, "rewind", sizeof("rewind") - 1)) {
235:         class_entry->iterator_funcs.if_valid = zend_hash_str_find_ptr(class_entry->function_table, "valid", sizeof("valid") - 1);
236:         class_entry->iterator_funcs.if_key = zend_hash_str_find_ptr(class_entry->function_table, "key", sizeof("key") - 1);
237:         class_entry->iterator_funcs.if_current = zend_hash_str_find_ptr(class_entry->function_table, "current", sizeof("current") - 1);
238:         class_entry->iterator_funcs.if_next = zend_hash_str_find_ptr(class_entry->function_table, "next", sizeof("next") - 1);
239:     }
240:
241:     if (inherited) {
242:         if (class_entry->iterator_funcs.if_rewind->common.scope != parent) {
243:             intern->flags |= SPL_FIXEDARRAY_OVERLOADED_REWIND;
244:         }
245:         if (class_entry->iterator_funcs.if_valid->common.scope != parent) {
246:             intern->flags |= SPL_FIXEDARRAY_OVERLOADED_VALID;
247:         }
248:         if (class_entry->iterator_funcs.if_key->common.scope != parent) {
249:             intern->flags |= SPL_FIXEDARRAY_OVERLOADED_KEY;
250:         }
251:         if (class_entry->iterator_funcs.if_current->common.scope != parent) {
252:             intern->flags |= SPL_FIXEDARRAY_OVERLOADED_CURRENT;
253:         }
254:         if (class_entry->iterator_funcs.if_next->common.scope != parent) {
255:             intern->flags |= SPL_FIXEDARRAY_OVERLOADED_NEXT;
256:         }
257:
258:         intern->fptr_offset_get = zend_hash_str_find_ptr(class_entry->function_table, "offsetGet", sizeof("offsetGet") - 1);
259:         if (intern->fptr_offset_get->common.scope == parent) {
260:             intern->fptr_offset_get = NULL;
261:         }
262:         intern->fptr_offset_set = zend_hash_str_find_ptr(class_entry->function_table, "offsetSet", sizeof("offsetSet") - 1);
263:         if (intern->fptr_offset_set->common.scope == parent) {
264:             intern->fptr_offset_set = NULL;
265:         }
266:         intern->fptr_offset_has = zend_hash_str_find_ptr(class_entry->function_table, "offsetExists", sizeof("offsetExists") - 1);
267:         if (intern->fptr_offset_has->common.scope == parent) {
268:             intern->fptr_offset_has = NULL;
269:         }
270:         intern->fptr_offset_del = zend_hash_str_find_ptr(class_entry->function_table, "offsetUnset", sizeof("offsetUnset") - 1);
271:         if (intern->fptr_offset_del->common.scope == parent) {
272:             intern->fptr_offset_del = NULL;
273:         }
274:         intern->fptr_count = zend_hash_str_find_ptr(class_entry->function_table, "count", sizeof("count") - 1);
275:         if (intern->fptr_count->common.scope == parent) {
276:             intern->fptr_count = NULL;
277:         }
278:     }
279:
280:     return intern->std;
281: }
282: /* }}} */
283:
284: static zend_object *spl_fixedarray_new(zend_class_entry *class_entry) /* {{{ */ {
285:     return spl_fixedarray_object_new(class_entry, NULL, 0);
286: }
287: /* }}} */
288:
289: static zend_object *spl_fixedarray_object_clone(zval *object) /* {{{ */ {
290:     zend_object *old_object;
291:     zend_object *new_object;
292:     old_object = Z_OBJ_P(object);
293:     new_object = spl_fixedarray_object_new(old_object->ce, object, 1);
294:     zend_objects_clone_members(new_object, old_object);
295:     return new_object;
296: }
297: /* }}} */
298:
299: static inline zval *spl_fixedarray_object_read_dimension_helper(spl_fixedarray_object *intern, zval *offset) /* {{{ */ {
300:     zend_long index;
301:
302:     /* we have to return NULL on error here to avoid memleak because of
303:      * the spl_uninitialized_zval_ptr */
304:     if (offset) {
305:         zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
306:         return NULL;
307:     }
308:
309:     if (IS_LONG_P(offset) && !IS_LONG(offset)) {
310:         index = spl_offset_convert_to_long(offset);
311:     } else {
312:         index = Z_LVAL_P(offset);
313:     }
314:
315:     if (index < 0 || (index >= intern->array.size)) {
316:         zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
317:         return NULL;
318:     }
319:     if (IS_UNDEF(intern->array.elements[index])) {
320:         return NULL;
321:     }
322:     if (IS_UNDEF(intern->array.elements[index])) {
323:         return NULL;
324:     }
325:     if (IS_UNDEF(intern->array.elements[index])) {
326:         return NULL;
327:     }
328:     if (IS_UNDEF(intern->array.elements[index])) {
329:         return NULL;
330:     }
331:     if (IS_UNDEF(intern->array.elements[index])) {
332:         return NULL;
333:     }
334:     if (IS_UNDEF(intern->array.elements[index])) {
335:         return NULL;
336:     }
337:     if (IS_UNDEF(intern->array.elements[index])) {
338:         return NULL;
339:     }
340:     if (IS_UNDEF(intern->array.elements[index])) {
341:         return NULL;
342:     }
343:     if (IS_UNDEF(intern->array.elements[index])) {
344:         return NULL;
345:     }
346:     if (IS_UNDEF(intern->array.elements[index])) {
347:         return NULL;
348:     }
349:     if (IS_UNDEF(intern->array.elements[index])) {
350:         return NULL;
351:     }
352:     if (IS_UNDEF(intern->array.elements[index])) {
353:         return NULL;
354:     }
355:     if (IS_UNDEF(intern->array.elements[index])) {
356:         return NULL;
357:     }
358:     if (IS_UNDEF(intern->array.elements[index])) {
359:         return NULL;
360:     }
361:     if (IS_UNDEF(intern->array.elements[index])) {
362:         return NULL;
363:     }
364:     if (IS_UNDEF(intern->array.elements[index])) {
365:         return NULL;
366:     }
367:     if (IS_UNDEF(intern->array.elements[index])) {
368:         return NULL;
369:     }
370:     if (IS_UNDEF(intern->array.elements[index])) {
371:         return NULL;
372:     }
373:     if (IS_UNDEF(intern->array.elements[index])) {
374:         return NULL;
375:     }
376:     if (IS_UNDEF(intern->array.elements[index])) {
377:         return NULL;
378:     }
379:     if (IS_UNDEF(intern->array.elements[index])) {
380:         return NULL;
381:     }
382:     if (IS_UNDEF(intern->array.elements[index])) {
383:         return NULL;
384:     }
385:     if (IS_UNDEF(intern->array.elements[index])) {
386:         return NULL;
387:     }
388:     if (IS_UNDEF(intern->array.elements[index])) {
389:         return NULL;
390:     }
391:     if (IS_UNDEF(intern->array.elements[index])) {
392:         return NULL;
393:     }
394:     if (IS_UNDEF(intern->array.elements[index])) {
395:         return NULL;
396:     }
397:     if (IS_UNDEF(intern->array.elements[index])) {
398:         return NULL;
399:     }
400:     if (IS_UNDEF(intern->array.elements[index])) {
401:         return NULL;
402:     }
403:     if (IS_UNDEF(intern->array.elements[index])) {
404:         return NULL;
405:     }
406:     if (IS_UNDEF(intern->array.elements[index])) {
407:         return NULL;
408:     }
409:     if (IS_UNDEF(intern->array.elements[index])) {
410:         return NULL;
411:     }
412:     if (IS_UNDEF(intern->array.elements[index])) {
413:         return NULL;
414:     }
415:     if (IS_UNDEF(intern->array.elements[index])) {
416:         return NULL;
417:     }
418:     if (IS_UNDEF(intern->array.elements[index])) {
419:         return NULL;
420:     }
421:     if (IS_UNDEF(intern->array.elements[index])) {
422:         return NULL;
423:     }
424:     if (IS_UNDEF(intern->array.elements[index])) {
425:         return NULL;
426:     }
427:     if (IS_UNDEF(intern->array.elements[index])) {
428:         return NULL;
429:     }
430:     if (IS_UNDEF(intern->array.elements[index])) {
431:         return NULL;
432:     }
433:     if (IS_UNDEF(intern->array.elements[index])) {
434:         return NULL;
435:     }
436:     if (IS_UNDEF(intern->array.elements[index])) {
437:         return NULL;
438:     }
439:     if (IS_UNDEF(intern->array.elements[index])) {
440:         return NULL;
441:     }
442:     if (IS_UNDEF(intern->array.elements[index])) {
443:         return NULL;
444:     }
445:     if (IS_UNDEF(intern->array.elements[index])) {
446:         return NULL;
447:     }
448:     if (IS_UNDEF(intern->array.elements[index])) {
449:         return NULL;
450:     }
451:     if (IS_UNDEF(intern->array.elements[index])) {
452:         return NULL;
453:     }
454:     if (IS_UNDEF(intern->array.elements[index])) {
455:         return NULL;
456:     }
457:     if (IS_UNDEF(intern->array.elements[index])) {
458:         return NULL;
459:     }
460:     if (IS_UNDEF(intern->array.elements[index])) {
461:         return NULL;
462:     }
463:     if (IS_UNDEF(intern->array.elements[index])) {
464:         return NULL;
465:     }
466:     if (IS_UNDEF(intern->array.elements[index])) {
467:         return NULL;
468:     }
469:     if (IS_UNDEF(intern->array.elements[index])) {
470:         return NULL;
471:     }
472:     if (IS_UNDEF(intern->array.elements[index])) {
473:         return NULL;
474:     }
475:     if (IS_UNDEF(intern->array.elements[index])) {
476:         return NULL;
477:     }
478:     if (IS_UNDEF(intern->array.elements[index])) {
479:         return NULL;
480:     }
481:     if (IS_UNDEF(intern->array.elements[index])) {
482:         return NULL;
483:     }
484:     if (IS_UNDEF(intern->array.elements[index])) {
485:         return NULL;
486:     }
487:     if (IS_UNDEF(intern->array.elements[index])) {
488:         return NULL;
489:     }
490:     if (IS_UNDEF(intern->array.elements[index])) {
491:         return NULL;
492:     }
493:     if (IS_UNDEF(intern->array.elements[index])) {
494:         return NULL;
495:     }
496:     if (IS_UNDEF(intern->array.elements[index])) {
497:         return NULL;
498:     }
499:     if (IS_UNDEF(intern->array.elements[index])) {
500:         return NULL;
501:     }
502:     if (IS_UNDEF(intern->array.elements[index])) {
503:         return NULL;
504:     }
505:     if (IS_UNDEF(intern->array.elements[index])) {
506:         return NULL;
507:     }
508:     if (IS_UNDEF(intern->array.elements[index])) {
509:         return NULL;
510:     }
511:     if (IS_UNDEF(intern->array.elements[index])) {
512:         return NULL;
513:     }
514:     if (IS_UNDEF(intern->array.elements[index])) {
515:         return NULL;
516:     }
517:     if (IS_UNDEF(intern->array.elements[index])) {
518:         return NULL;
519:     }
520:     if (IS_UNDEF(intern->array.elements[index])) {
521:         return NULL;
522:     }
523:     if (IS_UNDEF(intern->array.elements[index])) {
524:         return NULL;
525:     }
526:     if (IS_UNDEF(intern->array.elements[index])) {
527:         return NULL;
528:     }
529:     if (IS_UNDEF(intern->array.elements[index])) {
530:         return NULL;
531:     }
532:     if (IS_UNDEF(intern->array.elements[index])) {
533:         return NULL;
534:     }
535:     if (IS_UNDEF(intern->array.elements[index])) {
536:         return NULL;
537:     }
538:     if (IS_UNDEF(intern->array.elements[index])) {
539:         return NULL;
540:     }
541:     if (IS_UNDEF(intern->array.elements[index])) {
542:         return NULL;
543:     }
544:     if (IS_UNDEF(intern->array.elements[index])) {
545:         return NULL;
546:     }
547:     if (IS_UNDEF(intern->array.elements[index])) {
548:         return NULL;
549:     }
550:     if (IS_UNDEF(intern->array.elements[index])) {
551:         return NULL;
552:     }
553:     if (IS_UNDEF(intern->array.elements[index])) {
554:         return NULL;
555:     }
556:     if (IS_UNDEF(intern->array.elements[index])) {
557:         return NULL;
558:     }
559:     if (IS_UNDEF(intern->array.elements[index])) {
560:         return NULL;
561:     }
562:     if (IS_UNDEF(intern->array.elements[index])) {
563:         return NULL;
564:     }
565:     if (IS_UNDEF(intern->array.elements[index])) {
566:         return NULL;
567:     }
568:     if (IS_UNDEF(intern->array.elements[index])) {
569:         return NULL;
570:     }
571:     if (IS_UNDEF(intern->array.elements[index])) {
572:         return NULL;
573:     }
574:     if (IS_UNDEF(intern->array.elements[index])) {
575:         return NULL;
576:     }
577:     if (IS_UNDEF(intern->array.elements[index])) {
578:         return NULL;
579:     }
580:     if (IS_UNDEF(intern->array.elements[index])) {
581:         return NULL;
582:     }
583:     if (IS_UNDEF(intern->array.elements[index])) {
584:         return NULL;
585:     }
586:     if (IS_UNDEF(intern->array.elements[index])) {
587:         return NULL;
588:     }
589:     if (IS_UNDEF(intern->array.elements[index])) {
590:         return NULL;
591:     }
592:     if (IS_UNDEF(intern->array.elements[index])) {
593:         return NULL;
594:     }
595:     if (IS_UNDEF(intern->array.elements[index])) {
596:         return NULL;
597:     }
598:     if (IS_UNDEF(intern->array.elements[index])) {
599:         return NULL;
600:     }
601:     if (IS_UNDEF(intern->array.elements[index])) {
602:         return NULL;
603:     }
604:     if (IS_UNDEF(intern->array.elements[index])) {
605:         return NULL;
606:     }
607:     if (IS_UNDEF(intern->array.elements[index])) {
608:         return NULL;
609:     }
610:     if (IS_UNDEF(intern->array.elements[index])) {
611:         return NULL;
612:     }
613:     if (IS_UNDEF(intern->array.elements[index])) {
614:         return NULL;
615:     }
616:     if (IS_UNDEF(intern->array.elements[index])) {
617:         return NULL;
618:     }
619:     if (IS_UNDEF(intern->array.elements[index])) {
620:         return NULL;
621:     }
622:     if (IS_UNDEF(intern->array.elements[index])) {
623:         return NULL;
624:     }
625:     if (IS_UNDEF(intern->array.elements[index])) {
626:         return NULL;
627:     }
628:     if (IS_UNDEF(intern->array.elements[index])) {
629:         return NULL;
630:     }
631:     if (IS_UNDEF(intern->array.elements[index])) {
632:         return NULL;
633:     }
634:     if (IS_UNDEF(intern->array.elements[index])) {
635:         return NULL;
636:     }
637:     if (IS_UNDEF(intern->array.elements[index])) {
638:         return NULL;
639:     }
640:     if (IS_UNDEF(intern->array.elements[index])) {
641:         return NULL;
642:     }
643:     if (IS_UNDEF(intern->array.elements[index])) {
644:         return NULL;
645:     }
646:     if (IS_UNDEF(intern->array.elements[index])) {
647:         return NULL;
648:     }
649:     if (IS_UNDEF(intern->array.elements[index])) {
650:         return NULL;
651:     }
652:     if (IS_UNDEF(intern->array.elements[index])) {
653:         return NULL;
654:     }
655:     if (IS_UNDEF(intern->array.elements[index])) {
656:         return NULL;
657:     }
658:     if (IS_UNDEF(intern->array.elements[index])) {
659:         return NULL;
660:     }
661:     if (IS_UNDEF(intern->array.elements[index])) {
662:         return NULL;
663:     }
664:     if (IS_UNDEF(intern->array.elements[index])) {
665:         return NULL;
666:     }
667:     if (IS_UNDEF(intern->array.elements[index])) {
668:         return NULL;
669:     }
670:     if (IS_UNDEF(intern->array.elements[index])) {
671:         return NULL;
672:     }
673:     if (IS_UNDEF(intern->array.elements[index])) {
674:         return NULL;
675:     }
676:     if (IS_UNDEF(intern->array.elements[index])) {
677:         return NULL;
678:     }
679:     if (IS_UNDEF(intern->array.elements[index])) {
680:         return NULL;
681:     }
682:     if (IS_UNDEF(intern->array.elements[index])) {
683:         return NULL;
684:     }
685:     if (IS_UNDEF(intern->array.elements[index])) {
686:         return NULL;
687:     }
688:     if (IS_UNDEF(intern->array.elements[index])) {
689:         return NULL;
690:     }
691:     if (IS_UNDEF(intern->array.elements[index])) {
692:         return NULL;
693:     }
694:     if (IS_UNDEF(intern->array.elements[index])) {
695:         return NULL;
696:     }
697:     if (IS_UNDEF(intern->array.elements[index])) {
698:         return NULL;
699:     }
700:     if (IS_UNDEF(intern->array.elements[index])) {
701:         return NULL;
702:     }
703:     if (IS_UNDEF(intern->array.elements[index])) {
704:         return NULL;
705:     }
706:     if (IS_UNDEF(intern->array.elements[index])) {
707:         return NULL;
708:     }
709:     if (IS_UNDEF(intern->array.elements[index])) {
710:         return NULL;
711:     }
712:     if (IS_UNDEF(intern->array.elements[index])) {
713:         return NULL;
714:     }
715:     if (IS_UNDEF(intern->array.elements[index])) {
716:         return NULL;
717:     }
718:     if (IS_UNDEF(intern->array.elements[index])) {
719:         return NULL;
720:     }
721:     if (IS_UNDEF(intern->array.elements[index])) {
722:         return NULL;
723:     }
724:     if (IS_UNDEF(intern->array.elements[index])) {
725:         return NULL;
726:     }
727:     if (IS_UNDEF(intern->array.elements[index])) {
728:         return NULL;
729:     }
730:     if (IS_UNDEF(intern->array.elements[index])) {
731:         return NULL;
732:     }
733:     if (IS_UNDEF(intern->array.elements[index])) {
734:         return NULL;
735:     }
736:     if (IS_UNDEF(intern->array.elements[index])) {
737:         return NULL;
738:     }
739:     if (IS_UNDEF(intern->array.elements[index])) {
740:         return NULL;
741:     }
742:     if (IS_UNDEF(intern->array.elements[index])) {
743:         return NULL;
744:     }
745:     if (IS_UNDEF(intern->array.elements[index])) {
746:         return NULL;
747:     }
748:     if (IS_UNDEF(intern->array.elements[index])) {
749:         return NULL;
750:     }
751:     if (IS_UNDEF(intern->array.elements[index])) {
752:         return NULL;
753:     }
754:     if (IS_UNDEF(intern->array.elements[index])) {
755:         return NULL;
756:     }
757:     if (IS_UNDEF(intern->array.elements[index])) {
758:         return NULL;
759:     }
760:     if (IS_UNDEF(intern->array.elements[index])) {
761:         return NULL;
762:     }
763:     if (IS_UNDEF(intern->array.elements[index])) {
764:         return NULL;
765:     }
766:     if (IS_UNDEF(intern->array.elements[index])) {
767:         return NULL;
768:     }
769:     if (IS_UNDEF(intern->array.elements[index])) {
770:         return NULL;
771:     }
772:     if (IS_UNDEF(intern->array.elements[index])) {
773:         return NULL;
774:     }
775:     if (IS_UNDEF(intern->array.elements[index])) {
776:         return NULL;
777:     }
778:     if (IS_UNDEF(intern->array.elements[index])) {
779:         return NULL;
780:     }
781:     if (IS_UNDEF(intern->array.elements[index])) {
782:         return NULL;
783:     }
784:     if (IS_UNDEF(intern->array.elements[index])) {
785:         return NULL;
786:     }
787:     if (IS_UNDEF(intern->array.elements[index])) {
788:         return NULL;
789:     }
790:     if (IS_UNDEF(intern->array.elements[index])) {
791:         return NULL;
792:     }
793:     if (IS_UNDEF(intern->array.elements[index])) {
794:         return NULL;
795:     }
796:     if (IS_UNDEF(intern->array.elements[index])) {
797:         return NULL;
798:     }
799:     if (IS_UNDEF(intern->array.elements[index])) {
800:         return NULL;
801:     }
802:     if (IS_UNDEF(intern->array.elements[index])) {
803:         return NULL;
804:     }
805:     if (IS_UNDEF(intern->array.elements[index])) {
806:         return NULL;
807:     }
808:     if (IS_UNDEF(intern->array.elements[index])) {
809:         return NULL;
810:     }
811:     if (IS_UNDEF(intern->array.elements[index])) {
812:         return NULL;
813:     }
814:     if (IS_UNDEF(intern->array.elements[index])) {
815:         return NULL;
816:     }
817:     if (IS_UNDEF(intern->array.elements[index])) {
818:         return NULL;
819:     }
820:     if (IS_UNDEF(intern->array.elements[index])) {
821:         return NULL;
822:     }
823:     if (IS_UNDEF(intern->array.elements[index])) {
824:         return NULL;
825:     }
826:     if (IS_UNDEF(intern->array.elements[index])) {
827:         return NULL;
828:     }
829:     if (IS_UNDEF(intern->array.elements[index])) {
830:         return NULL;
831:     }
832:     if (IS_UNDEF(intern->array.elements[index])) {
833:         return NULL;
834:     }
835:     if (IS_UNDEF(intern->array.elements[index])) {
836:         return NULL;
837:     }
838:     if (IS_UNDEF(intern->array.elements[index])) {
839:         return NULL;
840:     }
841:     if (IS_UNDEF(intern->array.elements[index])) {
842:         return NULL;
843:     }
844:     if (IS_UNDEF(intern->array.elements[index])) {
845:         return NULL;
846:     }
847:     if (IS_UNDEF(intern->array.elements[index])) {
848:         return NULL;
849:     }
850:     if (IS_UNDEF(intern->array.elements[index])) {
851:         return NULL;
852:     }
853:     if (IS_UNDEF(intern->array.elements[index])) {
854:         return NULL;
855:     }
856:     if (IS_UNDEF(intern->array.elements[index])) {
857:         return NULL;
858:     }
859:     if (IS_UNDEF(intern->array.elements[index])) {
860:         return NULL;
861:     }
862:     if (IS_UNDEF(intern->array.elements[index])) {
863:         return NULL;
864:     }
865:     if (IS_UNDEF(intern->array.elements[index])) {
866:         return NULL;
867:     }
868:     if (IS_UNDEF(intern->array.elements[index])) {
869:         return NULL;
870:     }
871:     if (IS_UNDEF(intern->array.elements[index])) {
872:         return NULL;
873:     }
874:     if (IS_UNDEF(intern->array.elements[index])) {
875:         return NULL;
876:     }
877:     if (IS_UNDEF(intern->array.elements[index])) {
878:         return NULL;
879:     }
880:     if (IS_UNDEF(intern->array.elements[index])) {
881:         return NULL;
882:     }
883:     if (IS_UNDEF(intern->array.elements[index])) {
884:         return NULL;
885:     }
886:     if (IS_UNDEF(intern->array.elements[index])) {
887:         return NULL;
888:     }
889:     if (IS_UNDEF(intern->array.elements[index])) {
890:         return NULL;
891:     }
892:     if (IS_UNDEF(intern->array.elements[index])) {
893:         return NULL;
894:     }
895:     if (IS_UNDEF(intern->array.elements[index])) {
896:         return NULL;
897:     }
898:     if (IS_UNDEF(intern->array.elements[index])) {
899:         return NULL;
900:     }
901:     if (IS_UNDEF(intern->array.elements[index])) {
902:         return NULL;
903:     }
904:     if (IS_UNDEF(intern->array.elements[index])) {
905:         return NULL;
906:     }
907:     if (IS_UNDEF(intern->array.elements[index])) {
908:         return NULL;
909:     }
910:     if (IS_UNDEF(intern->array.elements[index])) {
911:         return NULL;
912:     }
913:     if (IS_UNDEF(intern->array.elements[index])) {
914:         return NULL;
915:     }
916:     if (IS_UNDEF(intern->array.elements[index])) {
917:         return NULL;
918:     }
919:     if (IS_UNDEF(intern->array.elements[index])) {
920:         return NULL;
921:     }
922:     if (IS_UNDEF(intern->array.elements[index])) {
923:         return NULL;
924:     }
925:     if (IS_UNDEF(intern->array.elements[index])) {
926:         return NULL;
927:     }
928:     if (IS_UNDEF(intern->array.elements[index])) {
929:         return NULL;
930:     }
931:     if (IS_UNDEF(intern->array.elements[index])) {
932:         return NULL;
933:     }
934:     if (IS_UNDEF(intern->array.elements[index])) {
935:         return NULL;
936:     }
937:     if (IS_UNDEF(intern->array.elements[index])) {
938:         return NULL;
939:     }
940:     if (IS_UNDEF(intern->array.elements[index])) {
941:         return NULL;
942:     }
943:     if (IS_UNDEF(intern->array.elements[index])) {
944:         return NULL;
945:     }
946:     if (IS_UNDEF(intern->array.elements[index])) {
947:         return NULL;
948:     }
949:     if (IS_UNDEF(intern->array.elements[index])) {
950:         return NULL;
951:     }
952:     if (IS_UNDEF(intern->array.elements[index])) {
953:         return NULL;
954:     }
955:     if (IS_UNDEF(intern->array.elements[index])) {
956:         return NULL;
957:     }
958:     if (IS_UNDEF(intern->array.elements[index])) {
959:         return NULL;
960:     }
961:     if (IS_UNDEF(intern->array.elements[index])) {
962:         return NULL;
963:     }
964:     if (IS_UNDEF(intern->array.elements[index])) {
965:         return NULL;
966:     }
967:     if (IS_UNDEF(intern->array.elements[index])) {
968:         return NULL;
969:     }
970:     if (IS_UNDEF(intern-&gt
```

```

377:     return spl_fixedarray_object_read_dimension_helper(intern, offset);
378: }
379: /* }}} */
380:
381:
382: static inline void spl_fixedarray_object_write_dimension_helper(spl_fixedarray_object *intern, zval *offset, zval *value) /* {{{ */
383: {
384:     zend_long index;
385:
386:     if (!offset) {
387:         /* '[array[] = value]' syntax is not supported */
388:         zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
389:         return;
390:     }
391:
392:     if (Z_TYPE_P(offset) != IS_LONG) {
393:         index = spl_offset_convert_to_long(offset);
394:     } else {
395:         index = Z_LVAL_P(offset);
396:     }
397:
398:     if (index < 0 || index >= intern->array.size) {
399:         zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
400:         return;
401:     } else {
402:         if (!IS_UNDEF(intern->array.elements[index])) {
403:             zval_ptr_dtor(&(intern->array.elements[index]));
404:         }
405:         ZVAL_DEREF(value);
406:         ZVAL_COPY(&(intern->array.elements[index]), value);
407:     }
408: }
409: /* }}} */
410:
411: static void spl_fixedarray_object_write_dimension(zval *object, zval *offset, zval *value) /* {{{ */
412: {
413:     spl_fixedarray_object *intern;
414:     zval tmp;
415:
416:     intern = Z_SPLFIXEDARRAY_P(object);
417:
418:     if (intern->fptr_offset_set) {
419:         if (!offset) {
420:             ZVAL_NULL(&tmp);
421:             offset = &tmp;
422:         } else {
423:             SEPARATE_ARG_IF_REF(offset);
424:         }
425:         SEPARATE_ARG_IF_REF(value);
426:         zend_call_method_with_2_params(object, intern->std.ce, intern->fptr_offset_set, "offsetSet", NULL, offset, value);
427:         zval_ptr_dtor(value);
428:         zval_ptr_dtor(offset);
429:         return;
430:     }
431:
432:     spl_fixedarray_object_write_dimension_helper(intern, offset, value);
433: }
434: /* }}} */
435:
436: static inline void spl_fixedarray_object_unset_dimension_helper(spl_fixedarray_object *intern, zval *offset) /* {{{ */
437: {
438:     zend_long index;
439:
440:     if (Z_TYPE_P(offset) != IS_LONG) {
441:         index = spl_offset_convert_to_long(offset);
442:     } else {
443:         index = Z_LVAL_P(offset);
444:     }
445:
446:     if (index < 0 || index >= intern->array.size) {
447:         zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
448:         return;
449:     } else {
450:         zval_ptr_dtor(&(intern->array.elements[index]));
451:         ZVAL_UNDEF(&(intern->array.elements[index]));
452:     }
453: }
454: /* }}} */
455:
456: static void spl_fixedarray_object_unset_dimension(zval *object, zval *offset) /* {{{ */
457: {
458:     spl_fixedarray_object *intern;
459:
460:     intern = Z_SPLFIXEDARRAY_P(object);
461:
462:     if (intern->fptr_offset_del) {
463:         SEPARATE_ARG_IF_REF(offset);
464:         zend_call_method_with_2_params(object, intern->std.ce, intern->fptr_offset_del, "offsetUnset", NULL, offset);
465:         zval_ptr_dtor(offset);
466:         return;
467:     }
468:
469:     spl_fixedarray_object_unset_dimension_helper(intern, offset);
470: }
471: }
472: /* }}} */
473:
474: static inline int spl_fixedarray_object_has_dimension_helper(spl_fixedarray_object *intern, zval *offset, int check_empty) /* {{{ */
475: {
476:     zend_long index;
477:     int retval;
478:
479:     if (Z_TYPE_P(offset) != IS_LONG) {
480:         index = spl_offset_convert_to_long(offset);
481:     } else {
482:         index = Z_LVAL_P(offset);
483:     }
484:
485:     if (index < 0 || index >= intern->array.size) {
486:         retval = 0;
487:     } else {
488:         if (!IS_UNDEF(intern->array.elements[index])) {
489:             retval = 1;
490:         } else if (check_empty) {
491:             if (zend_is_true(&(intern->array.elements[index]))) {
492:                 retval = 1;
493:             } else {
494:                 retval = 0;
495:             }
496:         } else /* != NULL and !check_empty */
497:             retval = 1;
498:     }
499: }
500:
501: return retval;
502: }
503: /* }}} */
504:
505: static int spl_fixedarray_object_has_dimension(zval *object, zval *offset, int check_empty) /* {{{ */
506: {
507:     spl_fixedarray_object *intern;
508:
509:     intern = Z_SPLFIXEDARRAY_P(object);
510:
511:     if (intern->fptr_offset_has) {
512:         zval rv;
513:         SEPARATE_ARG_IF_REF(offset);
514:         zend_call_method_with_2_params(object, intern->std.ce, intern->fptr_offset_has, "offsetExists", &rv, offset);
515:         zval_ptr_dtor(&rv);
516:         if (!IS_UNDEF(rv)) {
517:             zend_bool result = zend_is_true(rv);
518:             zval_ptr_dtor(&rv);
519:             return result;
520:         }
521:         return 0;
522:     }
523:
524:     return spl_fixedarray_object_has_dimension_helper(intern, offset, check_empty);
525: }
526: /* }}} */
527:
528: static int spl_fixedarray_object_count_elements(zval *object, zend_long *count) /* {{{ */
529: {
530:     spl_fixedarray_object *intern;
531:
532:     intern = Z_SPLFIXEDARRAY_P(object);
533:     if (intern->fptr_count) {
534:         zval rv;
535:         zend_call_method_with_0_params(object, intern->std.ce, intern->fptr_count, "count", &rv);
536:         if (!IS_UNDEF(rv)) {
537:             *count = zval_get_long(rv);
538:             zval_ptr_dtor(&rv);
539:         } else {
540:             *count = 0;
541:         }
542:     } else {
543:         *count = intern->array.size;
544:     }
545:     return SUCCESS;
546: }
547: /* }}} */
548:
549: /* {{{ proto void SplFixedArray::__construct([int size])
550: */
551: SPL_METHOD(SplFixedArray, __construct)
552: {
553:     zval *object = getThis();
554:     spl_fixedarray_object *intern;
555:     zend_long size = 0;
556:
557:     if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "|i", &size) == FAILURE) {
558:         return;
559:     }
560:
561:     if (size < 0) {
562:         zend_throw_exception(spl_ce_InvalidArgumentException, 0, "array size cannot be less than zero");
563:         return;
564:     }

```

```

565:     intern = Z_SPLFIXEDARRAY_P(object);
566:
567:     if (intern->array.size > 0) {
568:         /* called __construct() twice, bail out */
569:         return;
570:     }
571: }
572:
573: spl_fixedarray_init(&(intern->array), size);
574: }
575: /* }}} */
576:
577: /* {{{ proto void SplFixedArray::__wakeup()
578: */
579: SPL_METHOD(SplFixedArray, __wakeup)
580: {
581:     spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(getThis());
582:     HashTable *intern_ht = zend_std_get_properties(getThis());
583:     zval *data;
584:
585:     if (zend_parse_parameters_none() == FAILURE) {
586:         return;
587:     }
588:
589:     if (intern->array.size == 0) {
590:         int index = 0;
591:         int size = zend_hash_num_elements(intern_ht);
592:
593:         spl_fixedarray_init(&(intern->array), size);
594:
595:         ZEND_HASH_FOREACH_VAL(intern_ht, data) {
596:             ZVAL_COPY(&(intern->array.elements[index]), data);
597:             index++;
598:         } ZEND_HASH_FOREACH_END();
599:
600:         /* Remove the unserialized properties, since we now have the elements
601          * within the spl_fixedarray_object structure. */
602:         zend_hash_clean(intern_ht);
603:     }
604: }
605: /* }}} */
606:
607: /* {{{ proto int SplFixedArray::count(void)
608: */
609: SPL_METHOD(SplFixedArray, count)
610: {
611:     zval *object = getThis();
612:     spl_fixedarray_object *intern;
613:
614:     if (zend_parse_parameters_none() == FAILURE) {
615:         return;
616:     }
617:
618:     intern = Z_SPLFIXEDARRAY_P(object);
619:     return LONG(intern->array.size);
620: }
621: /* }}} */
622:
623: /* {{{ proto object SplFixedArray::toArray()
624: */
625: SPL_METHOD(SplFixedArray, toArray)
626: {
627:     spl_fixedarray_object *intern;
628:
629:     if (zend_parse_parameters_none() == FAILURE) {
630:         return;
631:     }
632:
633:     intern = Z_SPLFIXEDARRAY_P(getThis());
634:
635:     if (intern->array.size > 0) {
636:         int i = 0;
637:
638:         array_init(&(return_value));
639:         for (; i < intern->array.size; i++) {
640:             if (!IS_UNDEF(intern->array.elements[i])) {
641:                 zend_hash_index_update(&(Z_ARRVAL_P(return_value)), i, &(intern->array.elements[i]));
642:                 Z_TRY_ADDREF(intern->array.elements[i]);
643:             } else {
644:                 zend_hash_index_update(&(Z_ARRVAL_P(return_value)), i, &(uninitialized_zval));
645:             }
646:         }
647:     } else {
648:         ZVAL_EMPTY_ARRAY(&(return_value));
649:     }
650: }
651: /* }}} */
652:
653: /* {{{ proto object SplFixedArray::fromArray(array $data, bool $save_indexes)
654: */
655: SPL_METHOD(SplFixedArray, fromArray)
656: {
657:     zval *data;
658:     spl_fixedarray_array;
659:     spl_fixedarray_object *intern;
660:     int num;
661:     zend_bool save_indexes = 1;
662:
663:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "ab", &data, &save_indexes) == FAILURE) {
664:         return;
665:     }
666:
667:     num = zend_hash_num_elements(&(Z_ARRVAL_P(data)));
668:
669:     if (num > 0 & &save_indexes) {
670:         zval *element;
671:         zend_string *str_index;
672:         zend_long num_index, max_index = 0;
673:         zend_long tmp;
674:
675:         ZEND_HASH_FOREACH_KEY_VAL(&(Z_ARRVAL_P(data)), num_index, str_index) {
676:             if (str_index != NULL) { (zend_long)num_index < 0) {
677:                 zend_throw_exception(spl_ce_InvalidArgumentException, 0, "array must contain only positive integer keys");
678:                 return;
679:             }
680:
681:             if (num_index > max_index) {
682:                 max_index = num_index;
683:             }
684:         } ZEND_HASH_FOREACH_END();
685:
686:         tmp = max_index + 1;
687:         if (tmp <= 0) {
688:             zend_throw_exception(spl_ce_InvalidArgumentException, 0, "integer overflow detected");
689:             return;
690:         }
691:         spl_fixedarray_init(&(array), tmp);
692:
693:         ZEND_HASH_FOREACH_KEY_VAL(&(Z_ARRVAL_P(data)), num_index, str_index) {
694:             ZVAL_DEREF(element);
695:             ZVAL_COPY(&(array.elements[num_index]), element);
696:         } ZEND_HASH_FOREACH_END();
697:
698:     } else if (num > 0 & !&save_indexes) {
699:         zval *element;
700:         zend_long i = 0;
701:
702:         spl_fixedarray_init(&(array), num);
703:
704:         ZEND_HASH_FOREACH_VAL(&(Z_ARRVAL_P(data)), element) {
705:             ZVAL_DEREF(element);
706:             ZVAL_COPY(&(array.elements[i]), element);
707:             i++;
708:         } ZEND_HASH_FOREACH_END();
709:     } else {
710:         spl_fixedarray_init(&(array), 0);
711:     }
712:
713:     object_init_ex(&(return_value), spl_ce_SplFixedArray);
714:
715:     intern = Z_SPLFIXEDARRAY_P(&(return_value));
716:     intern->array = array;
717: }
718: /* }}} */
719:
720: /* {{{ proto int SplFixedArray::getSize(void)
721: */
722: SPL_METHOD(SplFixedArray, getSize)
723: {
724:     zval *object = getThis();
725:     spl_fixedarray_object *intern;
726:
727:     if (zend_parse_parameters_none() == FAILURE) {
728:         return;
729:     }
730:
731:     intern = Z_SPLFIXEDARRAY_P(object);
732:     return LONG(intern->array.size);
733: }
734: /* }}} */
735:
736: /* {{{ proto bool SplFixedArray::setSize(int size)
737: */
738: SPL_METHOD(SplFixedArray, setSize)
739: {
740:     zval *object = getThis();
741:     spl_fixedarray_object *intern;
742:     zend_long size;
743:
744:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "i", &size) == FAILURE) {
745:         return;
746:     }
747:
748:     if (size < 0) {
749:         zend_throw_exception(spl_ce_InvalidArgumentException, 0, "array size cannot be less than zero");
750:         return;
751:     }
752:

```

```

753: intern = _splfixedarray_p(object);
754:
755: spl_fixedarray_resize(intern->array, size);
756: RETURN_TRUE;
757: }
758: /* }}} */
759:
760: /* {{{ proto bool SplFixedArray::offsetExists(mixed $index)
761: Returns whether the requested $index exists. */
762: SPL_METHOD(SplFixedArray, offsetExists)
763: {
764:     zval *zindex;
765:     spl_fixedarray_object *intern;
766:
767:     IF_ZEND_PARSE_PARAMETERS(ZEND_NUM_ARGS(), "a", &zindex) == FAILURE {
768:         return;
769:     }
770:
771:     intern = _splfixedarray_p(getThis());
772:
773:     RETURN_BOOL(spl_fixedarray_object_has_dimension_helper(intern, zindex, 0));
774: } /* }}} */
775:
776: /* {{{ proto mixed SplFixedArray::offsetGet(mixed $index)
777: Returns the value at the specified $index. */
778: SPL_METHOD(SplFixedArray, offsetGet)
779: {
780:     zval *zindex, *value;
781:     spl_fixedarray_object *intern;
782:
783:     IF_ZEND_PARSE_PARAMETERS(ZEND_NUM_ARGS(), "a", &zindex) == FAILURE {
784:         return;
785:     }
786:
787:     intern = _splfixedarray_p(getThis());
788:     value = spl_fixedarray_object_read_dimension_helper(intern, zindex);
789:
790:     IF (value) {
791:         ZVAL_DEREF(value);
792:         ZVAL_COPY(return_value, value);
793:     } else {
794:         RETURN_NULL();
795:     }
796: } /* }}} */
797:
798: /* {{{ proto void SplFixedArray::offsetSet(mixed $index, mixed $newval)
799: Sets the value at the specified $index to $newval. */
800: SPL_METHOD(SplFixedArray, offsetSet)
801: {
802:     zval *zindex, *value;
803:     spl_fixedarray_object *intern;
804:
805:     IF_ZEND_PARSE_PARAMETERS(ZEND_NUM_ARGS(), "as", &zindex, &value) == FAILURE {
806:         return;
807:     }
808:
809:     intern = _splfixedarray_p(getThis());
810:     spl_fixedarray_object_write_dimension_helper(intern, zindex, value);
811:
812: } /* }}} */
813:
814: /* {{{ proto void SplFixedArray::offsetUnset(mixed $index)
815: Unsets the value at the specified $index. */
816: SPL_METHOD(SplFixedArray, offsetUnset)
817: {
818:     zval *zindex;
819:     spl_fixedarray_object *intern;
820:
821:     IF_ZEND_PARSE_PARAMETERS(ZEND_NUM_ARGS(), "a", &zindex) == FAILURE {
822:         return;
823:     }
824:
825:     intern = _splfixedarray_p(getThis());
826:     spl_fixedarray_object_unset_dimension_helper(intern, zindex);
827:
828: } /* }}} */
829:
830: static void spl_fixedarray_it_dtor(zend_object_iterator *iter) /* {{{ */
831: {
832:     spl_fixedarray_it *iterator = (spl_fixedarray_it *)iter;
833:
834:     zend_user_it_invalidate_current(iter);
835:     zval_ptr_dtor(&iterator->intern.it.data);
836: } /* }}} */
837:
838: static void spl_fixedarray_it_rewind(zend_object_iterator *iter) /* {{{ */
839: {
840:     spl_fixedarray_object *object = _splfixedarray_p(iterator->data);
841:
842:     IF (object->flags & SPL_FIXEDARRAY_OVERLOADED_REWIND) {
843:         zend_user_it_rewind(iter);
844:     } else {
845:         object->current = 0;
846:     }
847: } /* }}} */
848:
849: static int spl_fixedarray_it_valid(zend_object_iterator *iter) /* {{{ */
850: {
851:     spl_fixedarray_object *object = _splfixedarray_p(iterator->data);
852:
853:     IF (object->flags & SPL_FIXEDARRAY_OVERLOADED_VALID) {
854:         return zend_user_it_validate(iter);
855:     }
856:
857:     IF (object->current >= 0 && object->current < object->array.size) {
858:         return SUCCESS;
859:     }
860:     return FAILURE;
861: } /* }}} */
862:
863: static zval *spl_fixedarray_it_get_current_data(zend_object_iterator *iter) /* {{{ */
864: {
865:     zval *zindex;
866:     spl_fixedarray_object *object = _splfixedarray_p(iterator->data);
867:
868:     IF (object->flags & SPL_FIXEDARRAY_OVERLOADED_CURRENT) {
869:         return zend_user_it_get_current_data(iter);
870:     } else {
871:         zval *data;
872:
873:         ZVAL_LONG(&zindex, object->current);
874:
875:         data = spl_fixedarray_object_read_dimension_helper(object, &zindex);
876:         zval_ptr_dtor(&zindex);
877:
878:         IF (data == NULL) {
879:             data = &EG(uninitialized_zval);
880:         }
881:         return data;
882:     }
883: } /* }}} */
884:
885: static void spl_fixedarray_it_get_current_key(zend_object_iterator *iter, zval *key) /* {{{ */
886: {
887:     spl_fixedarray_object *object = _splfixedarray_p(iterator->data);
888:
889:     IF (object->flags & SPL_FIXEDARRAY_OVERLOADED_KEY) {
890:         zend_user_it_get_current_key(iter, key);
891:     } else {
892:         ZVAL_LONG(key, object->current);
893:     }
894: } /* }}} */
895:
896: static void spl_fixedarray_it_move_forward(zend_object_iterator *iter) /* {{{ */
897: {
898:     spl_fixedarray_object *object = _splfixedarray_p(iterator->data);
899:
900:     IF (object->flags & SPL_FIXEDARRAY_OVERLOADED_NEXT) {
901:         zend_user_it_move_forward(iter);
902:     } else {
903:         zend_user_it_invalidate_current(iter);
904:         object->current++;
905:     }
906: } /* }}} */
907:
908: /* {{{ proto int SplFixedArray::key()
909: Return current array key */
910: SPL_METHOD(SplFixedArray, key)
911: {
912:     spl_fixedarray_object *intern = _splfixedarray_p(getThis());
913:
914:     IF_ZEND_PARSE_PARAMETERS_NONE() == FAILURE {
915:         return;
916:     }
917:
918:     RETURN_LONG(intern->current);
919: } /* }}} */
920:
921: /* {{{ proto void SplFixedArray::next()
922: Move to next entry */
923: SPL_METHOD(SplFixedArray, next)
924: {
925:     spl_fixedarray_object *intern = _splfixedarray_p(getThis());
926:
927:     IF_ZEND_PARSE_PARAMETERS_NONE() == FAILURE {
928:         return;
929:     }
930:
931:     RETURN_LONG(intern->current++);
932: }

```

```

941: /* }}} */
942:
943: /* {{{ proto bool SplFixedArray::valid()
944: Check whether the datastructure contains more entries */
945: SPL_METHOD(SplFixedArray, valid)
946: {
947:     spl_fixedarray_object *intern = _splfixedarray_p(getThis());
948:
949:     IF_ZEND_PARSE_PARAMETERS_NONE() == FAILURE {
950:         return;
951:     }
952:
953:     RETURN_BOOL(intern->current >= 0 && intern->current < intern->array.size);
954: } /* }}} */
955:
956: /* {{{ proto void SplFixedArray::rewind()
957: Rewind the datastructure back to the start */
958: SPL_METHOD(SplFixedArray, rewind)
959: {
960:     spl_fixedarray_object *intern = _splfixedarray_p(getThis());
961:
962:     IF_ZEND_PARSE_PARAMETERS_NONE() == FAILURE {
963:         return;
964:     }
965:
966:     intern->current = 0;
967: } /* }}} */
968:
969: /* {{{ proto mixed SplFixedArray::current()
970: Return current datastructure entry */
971: SPL_METHOD(SplFixedArray, current)
972: {
973:     zval *zindex, *value;
974:     spl_fixedarray_object *intern = _splfixedarray_p(getThis());
975:
976:     IF_ZEND_PARSE_PARAMETERS_NONE() == FAILURE {
977:         return;
978:     }
979:
980:     ZVAL_LONG(&zindex, intern->current);
981:
982:     value = spl_fixedarray_object_read_dimension_helper(intern, &zindex);
983:
984:     zval_ptr_dtor(&zindex);
985:
986:     IF (value) {
987:         ZVAL_DEREF(value);
988:         ZVAL_COPY(return_value, value);
989:     } else {
990:         RETURN_NULL();
991:     }
992: } /* }}} */
993:
994: /* Iterator handler table */
995: static const zend_object_iterator_funcs spl_fixedarray_it_funcs = {
996:     spl_fixedarray_it_dtor,
997:     spl_fixedarray_it_valid,
998:     spl_fixedarray_it_get_current_data,
999:     spl_fixedarray_it_get_current_key,
1000:     spl_fixedarray_it_move_forward,
1001:     spl_fixedarray_it_rewind,
1002:     NULL
1003: };
1004:
1005: zend_object_iterator *spl_fixedarray_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
1006: {
1007:     spl_fixedarray_it *iterator;
1008:
1009:     IF (by_ref) {
1010:         zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1011:         return NULL;
1012:     }
1013:
1014:     iterator = emalloc(sizeof(spl_fixedarray_it));
1015:
1016:     zend_iterator_init(&zend_object_iterator*(iterator));
1017:
1018:     ZVAL_COPY(&iterator->intern.it.data, object);
1019:     iterator->intern.it.funcs = &spl_fixedarray_it_funcs;
1020:     iterator->intern.ce = ce;
1021:     ZVAL_UNDEF(&iterator->intern.value);
1022:
1023:     return iterator->intern.it;
1024: } /* }}} */
1025:
1026: /* {{{ */
1027:
1028: /* {{{ */
1029:
1030: ZEND_ARG_INFO_EX(arginfo_splfixedarray_construct, 0, 0, 0)
1031: ZEND_ARG_INFO(0, size)
1032: ZEND_ARG_INFO(0, offset)
1033:
1034: ZEND_ARG_INFO_EX(arginfo_splfixedarray_offsetGet, 0, 0, 1)
1035: ZEND_ARG_INFO(0, index)
1036: ZEND_ARG_INFO(0, data)
1037:
1038: ZEND_ARG_INFO_EX(arginfo_splfixedarray_offsetSet, 0, 0, 2)
1039: ZEND_ARG_INFO(0, index)
1040: ZEND_ARG_INFO(0, newval)
1041: ZEND_ARG_INFO(0, data)
1042:
1043: ZEND_ARG_INFO_EX(arginfo_splfixedarray_setSize, 0)
1044: ZEND_ARG_INFO(0, value)
1045: ZEND_ARG_INFO(0, data)
1046:
1047: ZEND_ARG_INFO_EX(arginfo_splfixedarray_fromArray, 0, 0, 1)
1048: ZEND_ARG_INFO(0, data)
1049: ZEND_ARG_INFO(0, save_indexes)
1050: ZEND_ARG_INFO(0, data)
1051:
1052: ZEND_ARG_INFO_EX(arginfo_splfixedarray_void, 0)
1053: ZEND_ARG_INFO(0, data)
1054:
1055: static const zend_function_entry spl_funcs_splfixedarray[] = { /* {{{ */
1056:     SPL_ME(splfixedarray, __construct, arginfo_splfixedarray_construct, ZEND_ACC_PUBLIC)
1057:     SPL_ME(splfixedarray, __wakeup, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1058:     SPL_ME(splfixedarray, __clone, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1059:     SPL_ME(splfixedarray, toArray, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1060:     SPL_ME(splfixedarray, fromArray, arginfo_splfixedarray_fromArray, ZEND_ACC_PUBLIC | ZEND_ACC_STATIC)
1061:     SPL_ME(splfixedarray, setSize, arginfo_splfixedarray_setSize, ZEND_ACC_PUBLIC)
1062:     SPL_ME(splfixedarray, setSize, arginfo_splfixedarray_setSize, ZEND_ACC_PUBLIC)
1063:     SPL_ME(splfixedarray, offsetExists, arginfo_splfixedarray_offsetGet, ZEND_ACC_PUBLIC)
1064:     SPL_ME(splfixedarray, offsetGet, arginfo_splfixedarray_offsetGet, ZEND_ACC_PUBLIC)
1065:     SPL_ME(splfixedarray, offsetSet, arginfo_splfixedarray_offsetSet, ZEND_ACC_PUBLIC)
1066:     SPL_ME(splfixedarray, offsetUnset, arginfo_splfixedarray_offsetGet, ZEND_ACC_PUBLIC)
1067:     SPL_ME(splfixedarray, rewind, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1068:     SPL_ME(splfixedarray, key, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1069:     SPL_ME(splfixedarray, next, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1070:     SPL_ME(splfixedarray, valid, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1071:     PHP_FE_END
1072: };
1073:
1074: /* {{{ PHP_MINIT_FUNCTION */
1075:
1076: /* {{{ PHP_MINIT_FUNCTION */
1077: PHP_MINIT_FUNCTION(spl_fixedarray)
1078: {
1079:     REGISTER_SPL_STM_CLASS_EX(splfixedarray, spl_fixedarray_new, spl_funcs_splfixedarray);
1080:     memory(spl_handler_splfixedarray, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1081:
1082:     spl_handler_splfixedarray.offset = &offsetOf(spl_fixedarray_object, scd);
1083:     spl_handler_splfixedarray.clone_obj = &spl_fixedarray_object_clone;
1084:     spl_handler_splfixedarray.read_dimension = &spl_fixedarray_object_read_dimension;
1085:     spl_handler_splfixedarray.write_dimension = &spl_fixedarray_object_write_dimension;
1086:     spl_handler_splfixedarray.unset_dimension = &spl_fixedarray_object_unset_dimension;
1087:     spl_handler_splfixedarray.has_dimension = &spl_fixedarray_object_has_dimension;
1088:     spl_handler_splfixedarray.count_elements = &spl_fixedarray_object_count_elements;
1089:     spl_handler_splfixedarray.get_properties = &spl_fixedarray_object_get_properties;
1090:     spl_handler_splfixedarray.get_gc = &spl_fixedarray_object_get_gc;
1091:     spl_handler_splfixedarray.dtor_obj = &zend_object_dtor_obj;
1092:     spl_handler_splfixedarray.free_obj = &spl_fixedarray_object_free_storage;
1093:
1094:     REGISTER_SPL_IMPLEMENT(splfixedarray, Iterator);
1095:     REGISTER_SPL_IMPLEMENT(splfixedarray, ArrayAccess);
1096:     REGISTER_SPL_IMPLEMENT(splfixedarray, Countable);
1097:
1098:     spl_ce_splfixedarray->get_iterator = &spl_fixedarray_get_iterator;
1099:
1100:     return SUCCESS;
1101: } /* }}} */
1102:
1103: /* {{{
1104:
1105: /* Local variables:
1106: * tab-width: 4
1107: * c-basic-offset: 4
1108: * End
1109: * vim600: noet sw=4 ts=4 fdm=marker
1110: * vim600: noet sw=4 ts=4
1111: */

```



```
1: /*
2:  *-----
3:  * | PHP Version 7 |
4:  *-----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  *-----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following uri: |
10:  * | http://www.php.net/license/3_01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  *-----
15:  * | Author: Antony Duvgal <trond@daylessday.org> |
16:  * | Elienne Kneuss <colder@php.net> |
17:  *-----
18:  */
19:
20: /* $Id$ */
21:
22: #ifndef SPL_FIXEDARRAY_H
23: #define SPL_FIXEDARRAY_H
24:
25: #extern PHPAPI zend_class_entry *spl_ce_SplFixedArray;
26:
27: #PHP_MINIT_FUNCTION(spl_fixedarray);
28:
29: #endif /* SPL_FIXEDARRAY_H */
30:
31: /*
32:  * Local Variables:
33:  * tab-width: 4
34:  * c-basic-offset: 4
35:  * End:
36:  * vim600: noet sw=4 ts=4 fM=marker
37:  * vim600: noet sw=4 ts=4
38:  */
```

```
1: /*
2:  *-----*
3:  * PHP Version 7
4:  *-----*
5:  * Copyright (c) 1997-2018 The PHP Group
6:  *-----*
7:  * This source file is subject to version 3.01 of the PHP license,
8:  * that is bundled with this package in the file LICENSE, and is
9:  * available through the world-wide-web at the following url:
10:  * http://www.php.net/license/3.01.txt
11:  * If you did not receive a copy of the PHP license and are unable to
12:  * obtain it through the world-wide-web, please send a note to
13:  * license@php.net so we can mail you a copy immediately.
14:  *-----*
15:  * Authors: Marcus Boerger <helly@php.net>
16:  *-----*
17:  */
18:
19: #ifndef PHP_SPL_H
20: #define PHP_SPL_H
21:
22: #include "php.h"
23: #include <stdint.h>
24:
25: #define PHP_SPL_VERSION PHP_VERSION
26:
27: #extern zend_module_entry spl_module_entry;
28: #define phpext_spl_ptr spl_module_entry
29:
30: #ifndef PHP_WIN32
31: # if defined SPL_EXPORTS
32: #  define SPL_API __declspec(dllexport)
33: # elif defined(COMPILE_DL_SPL)
34: #  define SPL_API __declspec(dllimport)
35: # else
36: #  define SPL_API /* nothing */
37: # endif
38: #elif defined(__GNUC__) && __GNUC__ >= 4
39: # define SPL_API __attribute__((visibility("default")))
40: #else
41: # define SPL_API
42: #endif
43:
44: #if defined(PHP_WIN32) && !defined(COMPILE_DL_SPL)
45: #undef phpext_spl
46: #define phpext_spl NULL
47: #endif
48:
49: #PHP_MINIT_FUNCTION(spl);
50: #PHP_MSHUTDOWN_FUNCTION(spl);
51: #PHP_RINIT_FUNCTION(spl);
52: #PHP_ASHUTDOWN_FUNCTION(spl);
53: #PHP_MINFO_FUNCTION(spl);
54:
55:
56: #END_BEGIN_MODULE_GLOBALS(spl)
57: zend_string *autoload_extensions;
58: HashTable *autoload_functions;
59: intptr_t hash_mask_handle;
60: intptr_t hash_mask_handlers;
61: int hash_mask_init;
62: int autoload_running;
63: #END_END_MODULE_GLOBALS(spl)
64:
65: #END_EXTERN_MODULE_GLOBALS(spl)
66: #define SPL_G(v) ZEND_MODULE_GLOBALS_ACCESSOR(spl, v)
67:
68: #PHP_FUNCTION(spl_classes);
69: #PHP_FUNCTION(class_parents);
70: #PHP_FUNCTION(class_implements);
71: #PHP_FUNCTION(class_used);
72:
73: #if PHPAPI zend_string *php_spl_object_hash(void *obj);
74:
75: #endif /* PHP_SPL_H */
76:
77: /*
78:  * Local Variables:
79:  * n-basis-offset: 4
80:  * tab-width: 4
81:  * End:
82:  * vim600: fdm=marker
83:  * vim: noet sw=4 ts=4
84:  */
```